TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
BoW	Bag of Words
CBF	Content-Based Filtering
CF	Collaborative Filtering
CI	Computational Intelligence
CR	Change Request
CVS	Control Version System
DAG	Direct Acyclic Graph
DCL	Dependency Constraint Language
DOI	Degree Of Interest
GA	Genetic Algorithm
IDE	Integrated Development Environment
KBF	Knowledge-Based Filtering
LSI	Latent Semantic Indexing
MIS	Method Invoking Sequence
NLP	Natural Language Processing
NN	Nearest Neighbors
QA	Question / Answer
RS	Recommendation System
RSSE	Recommendation System in Software Engineering

SCoReS	Source Code-based Recommendation System
SLR	Systematic Literature Review
TF-IDF	Term Frequency - Inverse Document Frequency

INTRODUCTION

Research Context and Problematic

As software systems evolve, their information resource including source code, external libraries and other documents (e.g. design documents) grow considerably as well. This diversity of information makes software systems' evolution and maintenance a challenging task. For instance, developers may encounter challenges in searching for relevant artifacts related to a given feature request or for source code examples and in adapting them to the programming task at hand. In some cases, documentation can be obsolete or not available which makes developers ask questions to more experienced developers who may be too busy to answer and then developers spend considerable time to get the desired information.

In this perspective, various information retrieval and research tools, usually based on regular expressions, have been developed to help developers locating information in which they are interested, but such tools are context-independent. So developers need to select interesting components according to the task at hand, which is usually a time consuming and an error-prone task. Therefore, Recommendation Systems in Software Engineering (RSSE) appear as interesting tools that consider the context of the developer's task in order to provide relevant information which ranges from project artifacts of the project under development (Cubranic et al., 2005) to components retrieved from the web (Sawadsky et al., 2013).

In literature, existing RSSEs come in different shapes and support various goals like recommending API method invocations (Long et al., 2009), reusable software components (McCarey et al., 2005), etc. However, they share many features and often require the same steps. A couple of surveys were conducted to analyze existing RSSEs (e.g. (Happel and Maalej, 2008), (Mohebzada et al., 2012)). Most of them focused on identifying when and what to recommend, while others tried to outline different recommendation techniques.

Also a handful works tried to highlight steps that are basic keys to implement RSSEs (e.g. (Robillard et al., 2010), (Mens and Lozano, 2014)). This process typically starts with the extraction of the context of the programming task at hand and provides an input to the recommendation engine which generates a set of recommendations that can be filtered before being presented to developers and maintainers. Figure 0.1 shows the basic components in RSSE.

In view of the aforementioned, we were interested to conduct a systematic review in order to deeply analyze a sample of existing RSSEs and to identify various features characterizing each component we need to build an RSSE.

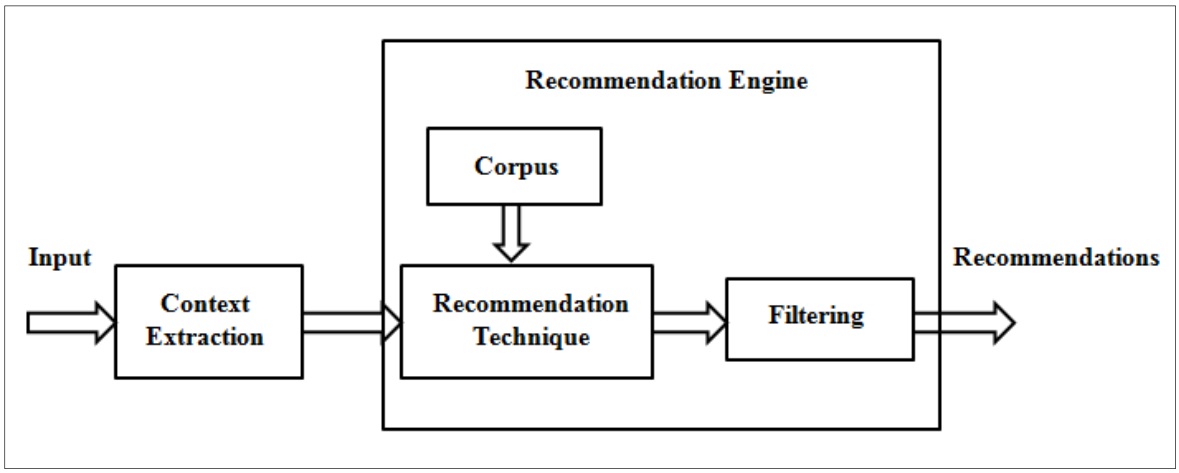

Figure 0.1 Building steps of an RSSE
Adapted from Maki et al. (2015, p.151)

Research objectives

The main goal of this research work is to examine the basic components for implementing RSSE tools and to identify their different design and implementation choices. In particular, we limit our study to RSSE tools supporting developers in maintenance and evolution tasks including refactoring, debugging, change tasks, etc.

To this end, the objectives specific to this thesis are:

- Identification of the main characteristics of the context extraction component
- Identification of the main characteristics of the recommendation engine component including its subcomponents: corpus, recommendation technique and filtering.

Thesis structure

The remainder of this thesis is organized as follows. The first chapter presents a literature review of relevant existing works on recommendation systems in general and in software engineering in particular. Chapter two describes the planning of our study by presenting our research questions and the selection criteria taken to answer these questions. Chapter three presents the description of the analyzed sample of RSSEs. The results of this analysis are presented in chapter four, and discussed in chapter five. Finally, we conclude our thesis and we present some future works.

CHAPTER 1

LITERATURE REVIEW

In this chapter, we present an overview of existing RS and RSSE surveys, and some relevant research works about RSSE building process and its evaluation. Finally, we identify the limitations that will be addressed by our review.

1.1 Basic concepts

In this section, we define the notion of recommendation systems in general. Then, we introduce the same concept in software engineering.

1.1.1 Recommendation System (RS)

An RS is a software tool providing useful suggestions about a particular item in order to help the user making a decision, for instance which book to buy or what music to listen to (Ricci et al., 2011). The need for RSs emerged especially with the introduction of e-commerce web sites and the explosive growth of information available on the web. In this perspective, an item denotes what the RS recommends to a user which can be a CD, a book, a movie, and so on. There are two types of recommendations: (1) personalized and (2) non-personalized recommendations (Ricci et al., 2011).

Personalized recommendations are usually presented as a ranked list of items. This ranking can be defined as a prediction of the most useful items. It is computed based on the user preferences. These preferences can be expressed:

- explicitly, e.g. as ratings assigned by the user for a particular item; or
- implicitly which are usually inferred by interpreting users' actions, e.g. visiting a particular item's page can be considered as an implicit sign of preference for that item.

Non-personalized recommendations are simpler to produce and are often used in magazines and newspapers, for instance, suggesting the top ten selections of magazines. This type of recommendations is not typically addressed by RS researches.

1.1.2 Recommendation System in Software Engineering (RSSE)

In software engineering, software development and maintenance activities present many information navigation issues and challenges. As software systems evolve, their information resources tend to keep growing (e.g. source code, change history, bug reports, discussion forums) and to depend on an ever-increasing set of external libraries.

Developers and maintainers usually tend to invoke existing source code components, e.g. reusable code snippets from project histories or methods from external libraries, rather than writing code from scratch. Thus, they may encounter two main challenges: (1) retrieving the suitable information from the various information resources, (2) learning its correct usage and adapting it to the programming task at hand. Therefore, this dynamicity and overwhelming diversity of information resources motivate the development of RSSE in order to support developers. Robillard et al. (2010) defined RSSE as « ... a software application that provides information items estimated to be valuable for a software engineering task in a given context. » (Robillard et al., 2010).

1.2 RS surveys

Many RS reviews have been conducted in order to present an overview of different existing recommendation techniques and to identify their drawbacks. In this section, we report some relevant RS reviews.

Adomavicius and Tuzhilin (2005) conducted a survey analyzing a sample of recommendation approaches in order to identify various limitations and to propose possible extensions. The analyzed approaches are classified into three categories: Content-Based Filtering (CBF), Collaborative Filtering (CF) and hybrid recommendation approaches.

CBF approach recommends items similar to the ones liked by the user in the past, based on his preferences and personal interests. These preferences can be presented as a set of keywords or categories. They are usually extracted from items descriptions which contain textual information, for instance, in a movies RS, preferences can be genres, lead actors, directors. A weighted measure can be affected to each keyword in order to determine its relevance in the textual description, e.g. Term Frequency - Inverse Document Frequency (TF-IDF) which is a weight assigned to every term according to its importance (frequency) to a document in a corpus or a collection of documents. In order to compute similarity, the recommendation techniques used can be based on heuristics, e.g. cosine similarity measure, or based on models using machine learning techniques, e.g. artificial neural networks. Adomavicius and Tuzhilin (2005) identified three main limitations in CBF approach:

- limited content analysis which is usually performed by a computer in the case of a textual content, otherwise it is performed manually which is often time-consuming;
- overspecialization as the recommended items are limited to items similar to those liked by the user and may include the same information; and
- new user problem as the new user did not yet rate items so the content-based approach would not be able to understand his preferences and, thus, to recommend relevant items.

CF approach recommends items which are most liked by users with similar preferences as the active user. For instance, in order to recommend a movie, CF approach tries to identify users with movies preferences similar to the ones of the active user and then recommends the movies which are most liked by the similar users previously identified. Similarity algorithms used by CF approaches are classified in two main categories: heuristic-based, e.g. Nearest Neighbors (NN) algorithm, and model-based algorithms, e.g. clustering. Adomavicius and Tuzhilin (2005) discussed the three following limitations in CF approach:

- new user problem which is identified in CBF limitations;
- new item problem as the new item cannot be recommended until it is rated by some users; and
- rating sparsity as the items that have been rated by few users would be rarely recommended. The same goes for users with different preferences; compared to the rest of users; who would not be able to have similar users and thus to get relevant recommendations. A possible solution to address this limitation is to use information incorporated in user profile in similarity computing.

Hybrid approach combines the CBF and CF approaches in order to address some of their limitations. This combination can be performed in four different ways:

- implementing CBF and CF techniques separately and then combining the two recommenders by combining their ratings into one final rating;
- adding some CBF characteristics to CF approach, for instance maintaining content-based users profiles in CF approach allows to address the ratings sparsity limitation as an unrated item could be recommended if it matches the user preferences;
- adding some CF characteristics to CBF approach, for instance performing the collaborative approach on a set of user profiles;
- implementing a single recommendation approach that incorporates CBF and CF characteristics.

Adomavicius and Tuzhilin (2005) proposed some extensions in order to avoid the limitations identified such as:

- better comprehension of users and items by using more advanced profiling techniques, e.g. data mining rules, instead of traditional techniques, e.g. keywords;
- incorporating multi-criteria ratings, e.g. restaurants recommendations may consider food, service and decor ratings;
- using implicit ratings, e.g. time spent visiting an item web page; and
- considering contextual information such as time (e.g. season, month, year), place, and companion in proposing travel-related recommendations.

The limitation of discarding contextual information has been addressed a couple years later by the same authors in (Adomavicius and Tuzhilin, 2011) where they discussed the concept of context in RSs and proposed three major approaches to incorporate context into the recommendation process. Adopting the representational view proposed by Dourish in (Dourish, 2004), Adomavicius and Tuzhilin (2011) considered context as a set of predefined information that doesn't change significantly over a short time period. This contextual information can be obtained in three different ways: (1) explicitly by asking questions, e.g. filling out a web form, (2) implicitly, e.g. the location of the user that can be detected by a mobile phone, or (3) by inferring, i.e. using data mining techniques. Adomavicius and Tuzhilin (2011) proposed the following three main paradigms to incorporate context in recommenders:

- pre-filtering that adds contextual information to the recommendation input, i.e. data is selected and constructed according to that specific context;
- post-filtering which adds contextual information to the recommendation output, i.e. ratings are computed on the entire data and then recommendations are filtered according to the contextual information of each user; and
- contextual modeling which adds contextual information to the recommendation function, i.e. context is used directly to compute ratings prediction.

In (Lu et al., 2015), the authors conducted a review of the latest RSs. The analyzed papers are classified into two main types: (1) papers on recommendation techniques, i.e. approaches and methods, and (2) papers on RS applications, i.e. software, which are clustered according to the application domains into eight main categories: e-tourism, e-business, e-government, e-commerce/e-shopping, e-learning, e-library, e-group activities and e-resource services. The reviewed recommendation techniques in (Lu et al., 2015) includes traditional techniques, e.g. CF and CBF (previously presented), Knowledge-Based Filtering (KBF), hybrid methods, and advanced techniques, e.g. Computational Intelligence (CI), social network-based, context awareness-based and group aggregation recommendation approaches. A short description of each of these techniques is presented in Table 1.1.

The review conducted in (Lu et al., 2015) shows that traditional recommendation techniques are still frequently used, in particular hybrid techniques which aim to avoid the limitations of using a single traditional recommendation technique, i.e. CF, CBF, KBF. Regarding advanced recommendation techniques, context aware and social network-based recommendation techniques are popular, and CI techniques are applied in all application domains. Yet, some open research topics have been identified in this review such as mobile-based context-sensitive and real time context awareness-based recommendation techniques. A summary of the reviewed RS applications in (Lu et al., 2015) is presented in Table 1.2.

Table 1.1 Summary of techniques' descriptions
Adapted from (Lu et al., 2015)

Approach	**Description**
KBF	This approach recommends items based on a deep knowledge about items (semantic knowledge, e.g. ontology). Depending on the user's preferences, KBF approach uses a set of constraints to describe which item has to be recommended
CI	This approach includes clustering techniques, artificial neural networks (ANN) and genetic algorithms. Clustering techniques gather similar items into one cluster and are usually used to find k-nearest neighbors. ANN technique is a weighted graph that links a set of inter-connected nodes. It is inspired by the architecture of the biological brain and it has been used to construct movies and TV recommender systems. Genetic algorithm (GA) is a stochastic search technique which is used often to address optimization problems.
Social network-based approaches	These approaches have emerged with the explosive growth of social networking tools. They help to overcome sparse data sets problem which is one of CF limitations (i.e. inability to find sufficient similar neighbors). They also improve the user's trust as the active user would be more influenced by suggestions from his friends than by website advertising.

Approach	Description
Context awareness-based approaches	These approaches use contextual information that could be relevant to recommend useful items in specific circumstances such as place, time, etc. Context is defined as « any information that can be used to characterize the situation of an entity. An entity could be a person, a place, or an object that is considered relevant to the interaction between a user and an application, including the user and the application themselves. » (Dey et al., 2001).
Group aggregation approaches	These approaches, known also as e-group activity, recommend a group of user suggestions when preferences of group members are unclear. They are applied in movies, music, events and travel plans recommendation.

Table 1.2 Summary of recommendation techniques in each application domain
Extracted from Lu et al. (2015, p.27)

Domains	Techniques								No. of listed references
	CBF	CF	KBF	Hybrid	Computational Intelligence	Social Network	Context Aware	Group Aggregation	
E-government	1	5	1	5	4				9
E-business		1	3	3	4				5
E-commerce	3	1	4	1	4	2			8
E-library	2	2		3	1				6
E-learning	2		11		2				10
E-tourism	5	9	9	9	3	2	11		18
E-resource	9	16	6	15	8	1	1		27
E-group activity	9	5	2	5	1			2	21
Total	**31**	**39**	**36**	**41**	**27**	**6**	**12**	**2**	**104**

1.3 Relevant RSSE works

In this section, we present an overview of relevant works on RSSEs including surveys and other works that focus on building and evaluating RSSEs.

1.3.1 RSSE surveys

Compared to RS works, only a handful reviews have been conducted on RSSE. In this section, we report relevant RSSE reviews published in the last decade.

In (Happel and Maalej, 2008), the authors conducted a survey of the papers that were published between 2003 and 2008 (six RSSEs). This survey aims to identify potentials and limitations of the analyzed RSSEs with a particular focus on their architecture (e.g. client/ server, web application), their trigger events (proactive or reactive) and the type of recommended information (e.g. methods, project artifacts). Happel and Maalej (2008) outlined a recommendation landscape following two main dimensions:

- when to recommend, i.e. recommendation process is triggered proactively ("Propose") or reactively ("Ask to share"); and
- what to recommend, i.e. information to recommend which is classified into development information (e.g. code, project artifacts) and collaboration information (e.g. people to contact).

Table 1.3 summarizes the outlined recommendation landscape. Then, the authors identified some limitations such as: (1) the non-flexible architecture (e.g. standalone applications), and (2) the disregard of contextual information of the programming task at hand, or simply its restriction to the file level.

In the same perspective, the authors in (Mohebzada et al., 2012) attempted to identify research gaps in RSSE by providing an overview of recommendation systems for requirements engineering.

This study used a systematic mapping that included 23 publications between 2004 and 2011. The authors outlined the following characteristics of recommendation systems for requirements engineering:

- recommendation techniques (e.g. CF, CBF, etc.);
- types of recommended items (e.g. stakeholders of a project, software project planning, etc.);
- recommendation modes (i.e. proactive or reactive) and the output form (e.g. web page);
- cross-dimensional features (e.g. user's feedback, rationale behind recommendations, etc.);
- recommender architecture (e.g. web-based tool, standalone desktop application ,etc.).

Table 1.3 Recommendation landscape
Extracted from Happel and Maalej (2008, p.13)

What / **When**		**Information Access** **Propose...**	**Information Provision** **Ask to share...**
Development	Code	Auto completion, code examples, methods to use	Ways of reusing APIs, used documentations
	Artifacts	Related, useful artifacts	Artifacts used for solving a specific problem
	Quality measures	Problematic change, Patterns to improve quality	How problems have been solved, new patterns
	Tools	Not used features, How-to automate specific tasks	Experience reports on using new tools
Collaboration	People	Experts to contact	Associations of people with expertise areas
	Awareness measures	Ad-hoc collaboration	Collaboration artifacts (mail, chat, decision rationale)
	Status Priorities	Open related issues, Risks New priorities	Status, open issues Reason of priority changes

The analysis of Mohebzada et al. (2012) revealed some limitations of recommendation systems for requirements engineering such as:

- non-integrated recommenders in existing work environments of requirements engineering which could be addressed by plug-ins architecture;
- limited "explainability features" like rationale behind recommendations; and
- limited number of proactive recommenders.

More recently, in (Pakdeetrakulwong et al., 2014), the authors analyzed a sample of 25 RSSEs that were published between 2006 et 2014. The reviewed papers were classified according to the software development life cycle phases as follows: (1) nine requirements, gathering and analysis recommenders, (2) three design recommenders, (3) eleven implementation recommenders, and only (4) two testing recommenders. This study outlined the different recommendation techniques and the knowledge representation used (e.g. semantic representation) according to recommender goals and identified benefits and issues of the existing RSSEs. The authors noticed that the analyzed recommenders were developed to improve software productivity only for one software development phase and in particular the implementation phase. Thus, it would be of great help if recommenders help software teams in more than one phase of software life cycle. Regarding recommendation techniques and knowledge representations, the traditional representations based on structured or semi-structured format of data and the syntactic matching operations are the most used ones. These features can be improved by leveraging semantic ontologies.

1.3.2 Building RSSE works

In (Robillard et al., 2010), the authors presented an overview of some relevant recommenders focusing on how RSSEs can help developers. Also they outlined some design dimensions, potentials and gaps of existing RSSEs. In this study, the main dimensions identified (summarized in Table 1.4) involve:

- a data context collection process which can be implicit or explicit and mainly involve information like the user's past interactions (e.g. browsed components, etc.) and the current task (e.g. debugging, adding new feature, etc.);
- a recommendation engine that analyze additional data to generate recommendations using ranking techniques; and
- a user interface that triggers the recommendation process implicitly or explicitly (i.e. proactive and reactive modes) and presents results to the user.

This overview revealed some RSSE benefits like proactive mode which delivers automatically relevant information to developers rather than waiting for an explicit request. However, many limitations have been identified such as: “cold-start problem” of a project that could be addressed by leveraging data from other similar projects, and the output form which is presented as a list of recommendations in most of RSSEs so there are a limited explanation features.

Table 1.4 RSSE design dimensions
Extracted from Robillard et al. (2010, p.85)

<table>
<tr><th>Nature of the context</th><th>Recommendation engine</th><th>Output mode</th></tr>
<tr><td rowspan="3">Input:
explicit | implicit | hybrid</td><td>Data:
source | change | bug reports | mailing lists | interaction history | peers' actions</td><td>Mode:
push | pull</td></tr>
<tr><td>Ranking:
yes | no</td><td>Presentation:
batch | inline</td></tr>
<tr><td colspan="2">Explanations:
from none to detailed</td></tr>
<tr><td colspan="3">User feedback:
None | locally adjustable | individually adaptive | globally adaptive</td></tr>
</table>

In the same perspective, the authors in (Robillard and Walker, 2014) reviewed the various software information sources that could be relevant to generate recommendations (e.g. project history, external libraries, user interaction traces, etc.), and presented a more detailed overview of RSSEs aspects. These aspects are:

- data preprocessing such as parsing source code or analyzing commits;
- capturing context by gathering all information about the current development task;
- generating recommendations by performing a recommendation technique that takes as an input the processed data and the captured context; and
- presenting recommendations to the developer.

This general overview has been detailed by Mens and Lozano particularly for Source Code-based Recommendation System (SCoReS) in (Mens and Lozano, 2014) in order to outline relevant decisions to build a SCoReS. First, the authors presented an overview of a handful of existing recommenders, and then tackled important development choices of a SCoReS. These choices were classified according to the phase of the development cycle into two main categories: (1) decisions related to the recommendation approach, and (2) decisions related to the user interactions with the recommender. Table 1.5 summarizes these decisions. Also, the authors tried to answer these key decisions by going through a sample of SCoRes and discussing the development choices taken.

Table 1.5 Kinds of development decisions to be taken when building a SCoReS
Extracted from Mens and Lozano (2014, p.103)

	Requirements	**Design**	**Implementation**	**Validation**
Approach	1. Intent	3. Corpus	5. Method	7. Support
User interaction	2. HCI	4. General I/O	6. Detailed I/O	8. Interaction

In table 1.6, we detail the proposed decisions related to the approach and to the user interaction in each development cycle phase.

Table 1.6 Detailed development decisions of a SCoReS
Adapted from Mens and Lozano (2014, p.120-121)

	Requirements	**Design**	**Implementation**	**Validation**
Approach	1. Intent: • Intended user • Supported task • Cognitive support • Proposed information	3. Corpus • Program code • Complementary information • Correlated information	5. Method • Data selection • Type of analysis • Data requirements • Intermediate representation • Analysis technique • Filtering	7. Support • Empirical validation • Usefulness • Correctness
User interaction	2. HCI (Human Computer Interaction) • Type of the system • Type of recommender • User involvement	4. General Input/Output • Input mechanism • Nature of input • Response triggers • Nature of output • Type of output	6. Detailed Input/Output • Type of input • Multiplicity of output	8. Interaction • Usability • System availability • Data availability

More recently in (Proksch et al., 2015), the authors outline the different steps that should be taken to build a RSSE using some examples of recommenders that assist developers in API usage. The identified steps are described as follows:

1. Framing the problem to solve by determining the task (i.e. the goal), the context (i.e. information and tool environment) and the target user (e.g. novice developers);
2. Determining the inputs, i.e. data available to provide recommendations (e.g. open-source repositories, QA websites, etc.);
3. Building the recommender, i.e. mechanisms used to generate recommendations using inputs, including traditional techniques (e.g. collaborative filtering), data mining and machine learning(e.g. association rule mining);
4. Delivering recommendations which includes the recommendation mode (reactive or proactive) and the presentation of the provided recommendations; and
5. Evaluating the recommender including the evaluation of the proposals presentation (e.g. creating mock-ups in early design stages) and the recommendation engine (e.g. user studies, automated experiments, etc.).

1.3.3 Evaluating RSSE works

As evaluation is an important step to complete the building process of any software tool, we present an overview of some relevant works on approaches and metrics to evaluate RSSEs.

In (Avazpour et al., 2014), the authors review a range of evaluation metrics, measures and commonly used approaches to evaluate RSSEs. As a first step, the authors investigate a set of dimensions that can be relevant to assess RSSE quality. These dimensions are grouped into four main categories (summarized in Table 1.7):

- Recommendation-centric dimensions which evaluate the generated recommendations;
- User-centric dimensions assess the degree to which RSSE fulfills the user needs;
- System-centric dimensions evaluate the recommendation system itself; and
- Delivery-centric dimensions gauge the recommendation system in the context of use.

Table 1.7 Categorization of dimensions
Extracted from Avazpour et al. (2014, p.247)

Recommendation-centric	**User-centric**	**System-centric**	**Delivery-centric**
Correctness	Trustworthiness	Robustness	Usability
Coverage	Novelty	Learning rate	User preference
Diversity	Serendipity	Scalability	
Recommender confidence	Utility	Stability	
	Risk	Privacy	

For each dimension, the authors outline the most commonly used metrics. For instance, correctness which evaluates how close the provided recommendations are to users' interests, can be differently measured depending on the type of the generated recommendations (e.g. predicting user ratings, ranking items, etc). For instance, to recommend interesting items, classification metrics such as precision, recall, accuracy and specificity are the most commonly used metrics. Table 1.8 summarizes the identified metrics and techniques.

Table 1.8 Summary of metrics
Extracted from Avazpour et al. (2014, p.267)

Dimension	**Metric / Technique**	**Type(s)**
Correctness	*Ratings*: root-mean-square-error, normalized (RMSE), mean absolute error (MAE), normalized MAE *Ranking*: normalized distance-based performance measure, Spearman's ρ, Kendall's τ, normalized discounted cumulative gain *Classification*: precision, recall, false positive rate, specificity, F-measure, receiver operating characteristic curve	Quantitative

Dimension	Metric / Technique	Type(s)
Coverage	catalog coverage, weighted catalog coverage, prediction coverage, weighted prediction coverage	Quantitative
Diversity	diversity measure, relative diversity, precision–diversity curve, Q-statistics, set theoretic difference of recommendation lists	Quantitative
Trustworthiness	user studies	Qualitative
Confidence	neighborhood-aware similarity model, similarity indicators	Qualitative / Quantitative
Novelty	comparison of recommendation lists and user profiles, counting popular items	Qualitative / Quantitative
Serendipity	comparison of recommendation lists and user profiles, ratability	Qualitative / Quantitative
Utility	profit-based utility function, study user intention, user studies	Qualitative / Quantitative
Risk	depends on application and user preference	Qualitative
Robustness	prediction shift, average hit ratio, average rank	Quantitative
Learning rate	correctness over time	Quantitative
Usability	user studies (survey, observation, monitoring)	Qualitative / Quantitative
Scalability	training time, recommendation throughput	Quantitative
Stability	prediction shift	Quantitative
Privacy	differential privacy, RMSE vs. differential privacy curve	Qualitative / Quantitative
User preference	user studies	Qualitative / Quantitative

As an example of evaluation techniques, the authors in (Said et al., 2014) outline the concepts of benchmarking process for evaluating RSSEs and present a multi-dimensional approach.

Benchmarking is an evaluation methodology used to assess the quality of a tool in comparison with other tools, and its process follows four main steps: (1) target specification, (2) data collection, (3) evaluation and analysis, and (4) implementation. Traditional benchmarking techniques usually evaluate one dimension (e.g. accuracy) and neglect business and technical aspects. The proposed multi-dimensional approach is composed of three main aspects:

1. User aspects which are related to the impact on the user's attitude (e.g. persuasiveness, trustworthiness) and aim to identify interesting items and to reduce the time of the decision-making process;
2. Business aspects which are related to business requirements such as increased user retention and user loyalty and can be measured using metrics like click through-rate (the ratio of recommendation selected by the user per the total number of recommendations) and average page views per visit in case of websites; and
3. Technical aspects which consider technical constraints including data, system (hardware and software limitations), reactivity (ability to provide relevant recommendations in real-time), scalability (ability to provide relevant recommendations independently of the dataset size), robustness, etc.

1.4 Limitations of existing works

In this section, we summarize the limitations identified in both RSSE surveys and works that build RSSEs.

Regarding RSSE surveys, we noticed that existing surveys aim to identify potentials and gaps of RSSE with a particular focus on some features like the recommender’s architecture (plug-ins, web applications, etc.), its mode (reactive or proactive), the recommendation techniques and the nature of information presented to the developer. However, these surveys did not discuss the input nature, i.e. information to be extracted that can be relevant to the task at hand, and the data used to provide recommendations, and whether this data is processed before being used to generate recommendations.

The same features (i.e. recommender modes, recommendation technique and output) were considered by works that build RSSEs. In particular, the work by Mens and Lozano highlights more features to consider in each development life cycle phase of building source code-based recommenders. However, we tried to use their classification on some RSSEs with inputs different from source code (e.g. queries, project artifacts, etc.) and it ran into many ambiguities, such as:

- the ambiguity of recommender types which was defined as advisor, finder or validator;
- the user involvement and the input mechanism can be considered as one decision that explain how the input should be entered.

We noticed also that the input type which requires whether an additional information has to be selected or not, is requested in the implementation phase in the development cycle. However, it should be considered in earlier phases. These ambiguities forced us to consider a different approach.

In particular, we concluded that there was a need to clarify the key steps and to characterize the basic components of an RSSE regardless of the input type and the supporting goal. However, we do not focus on RSSE evaluation means in this survey.

CHAPTER 2

RESEARCH METHODOLOGY

To conduct our systematic literature review, we adopted the approach proposed in (Kitchenham, 2004). This approach is used to identify and evaluate available research in a particular field and includes three main phases:

- Planning of the study by developing a review protocol that includes the research questions, the search strategy, the selection criteria and the data extraction strategy;
- Executing the study includes the collection of data addressing the research questions; and
- Analyzing / interpreting the results of the study.

In this chapter, we detail the first step to conduct systematic literature review. The other steps are covered by the following chapters.

2.1 Planning of the study

In this section, we present the planning of the study we used to identify the set of papers and to filter out papers we consider relevant to our research questions. Figure 2.1 depicts our research methodology and details in particular the planning of our study.

2.1.1 Research questions

In this study, we formulated the following research questions:

- (Q1) : Which features characterize the context extraction process adopted by RSSEs ?
- (Q2) : Which features characterize the recommendation engine used by RSSEs to provide recommendations ?

In order to answer these research questions, we selected a sample of RSSE papers published in scientific conferences and journals within the last decade.

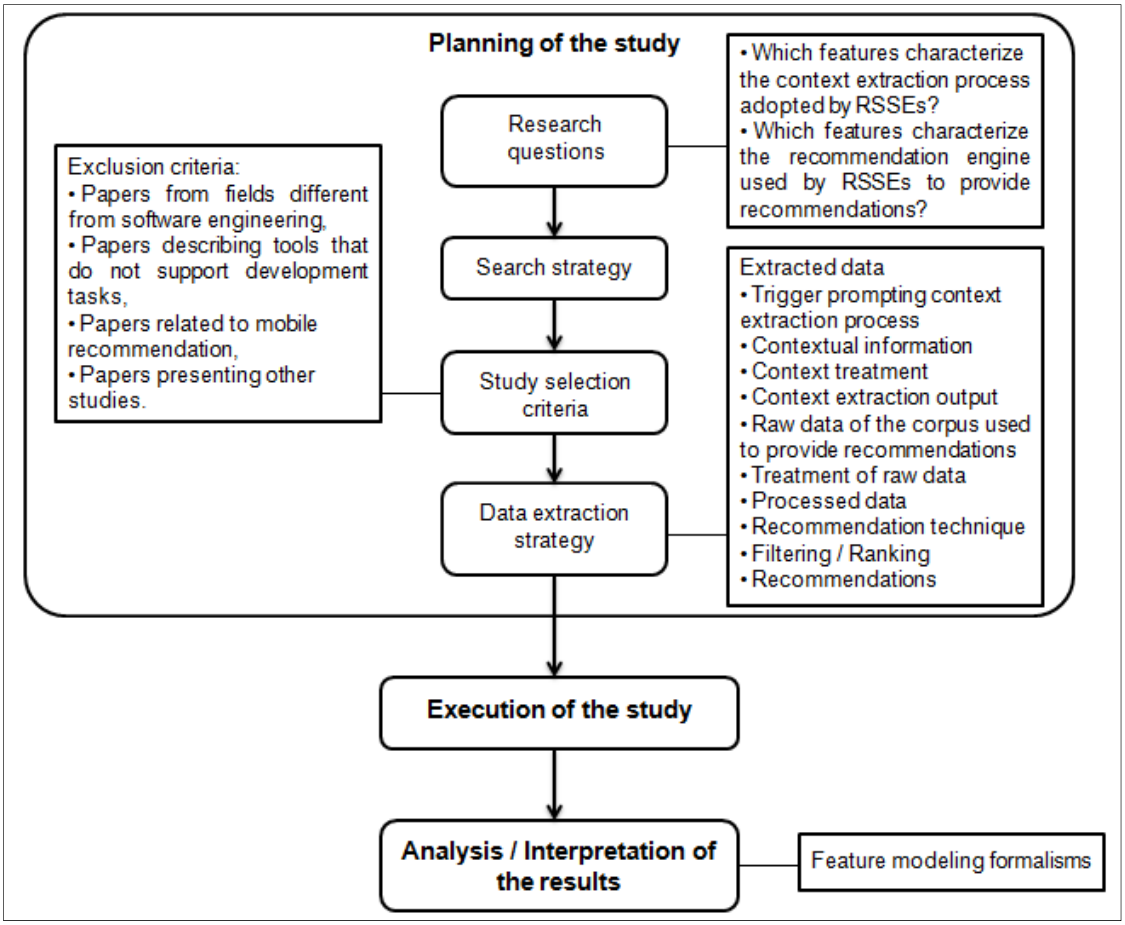

Figure 2.1 Research methodology

2.1.2 Search strategy

To select a sample of papers published in scientific conferences and journals within the last decade, we queried the research engine Compendex, known also as Engineering Village, which is an engineering bibliographic database providing a searchable index of the scientific literature. We performed the following query:

- (("recommendation system" OR "recommendation tool" OR "recommender") AND ("software engineering" OR "software development" OR "software maintenance" OR "software evolution" OR "software project")).

The search related to this query returned 272 results on which we performed a preliminary screening based on papers' title and abstract. Based on our selection criteria, described in the next subsection, we retained 78 publications that contain three duplicate tools (two different papers of Rascal and NavClus). We performed a second screening based on papers' introduction and conclusion and we filtered out 21 paper. After reading the rest of papers, we kept 36 relevant papers. Figure 2.2 summarizes the filtering approach.

To enlarge our study, we selected RSSEs that were analyzed in (Mens and Lozano, 2014). We kept papers that are published within the last decade and we filtered out those that were found by our query (Mendel (Lozano et al., 2011), Hipikat (Cubranic et al., 2005), Strathcona (Holmes et al., 2006)). We retained a sample of 10 publications. The final sample of the analyzed RSSEs is composed of 46 relevant papers.

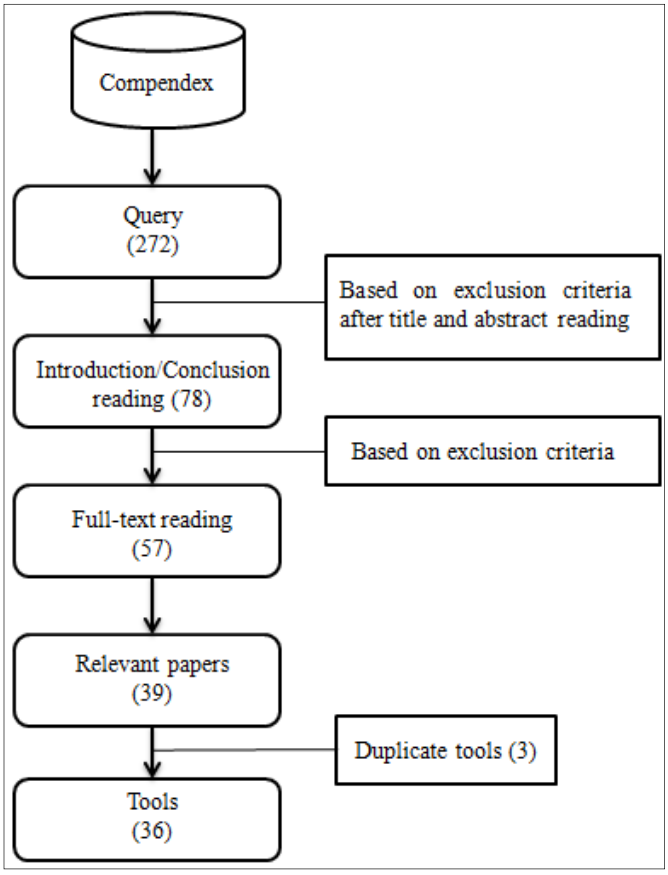

Figure 2.2 Filtering approach

2.1.3 Selection criteria

We were interested only in the publications presenting recommendation systems in software engineering, particularly the ones related to software development. To do so, we defined the following exclusion criteria:

i. Papers from fields other than software engineering, e.g. e-commerce, e-tourism, e-business, e-learning, health-care, etc.
ii. Papers presenting tools that do not support software development tasks, e.g. tools supporting requirements elicitation, works management (e.g. helping managers building their teams), or evaluation of recommendation systems.
iii. Papers presenting other studies which were discussed in the previous chapter (e.g. literature reviews).

2.1.4 Data extraction strategy

We prepared tables to accurately record any information that can be relevant to answer our research questions. To do so, we followed two main steps in order to extract the following data categories:

1. Context extraction phase which is usually prompted by a triggering event to collect the contextual information (i.e. input) that may be treated in order to generate an output. The context input has to be retrieved within a given scope and possibly involves the extraction of specific elements. Figure 2.3 shows an overview of this phase.
2. The recommendation engine takes as an input the context extraction output and then performs a treatment in order to generate recommendations which can be filtered or ranked before being presented to the developer. The recommendation engine possibly involves a corpus which is a set of raw data that can be treated in order to generate an output (processed data) used to recommend items. An overview of this phase is shown in Figure 2.4.

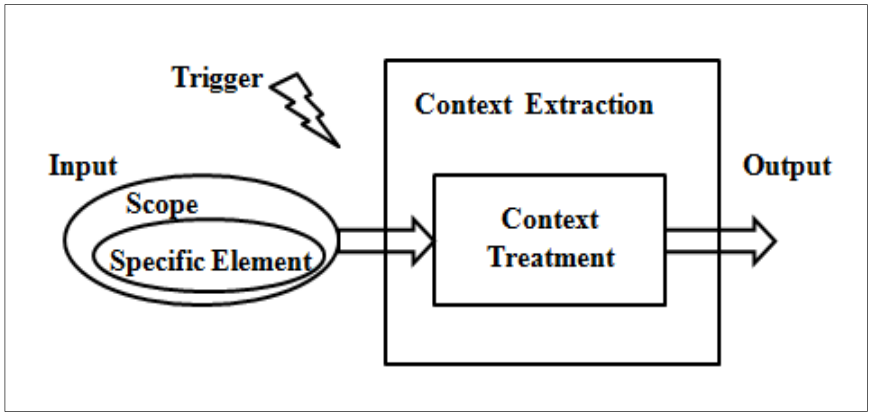

Figure 2.3 Context extraction phase

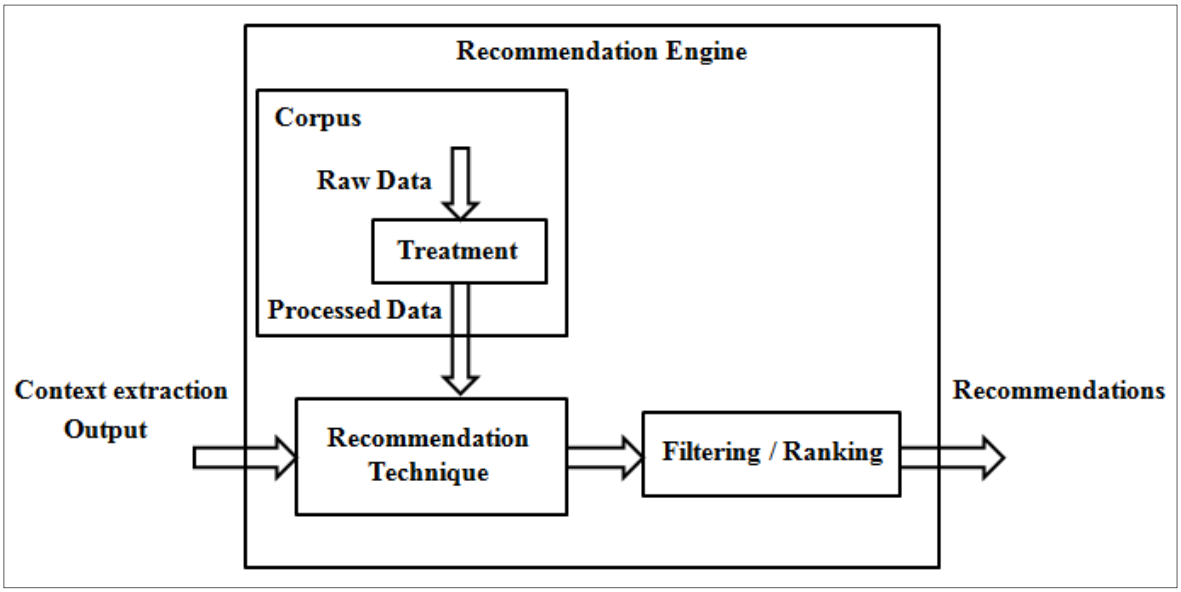

Figure 2.4 Recommendation Engine

CHAPTER 3

EXECUTION OF THE STUDY

In this chapter, we present the sample of RSSEs analyzed. We cluster these tools according to the supported goal into seven main categories. Table 3.1 presents an overview of this categorization.

Table 3.1 Categorization of analyzed tools

Supported goal	**Number of papers**
Change tasks	12
API usage	6
Refactoring	4
Debugging and testing	11
Reusable software components	5
Exploring codebases	5
Others: Prototyping Tagging Recommending experts	 1 1 1

In some categories, we describe a sample of the selected tools. The remaining recommenders are described in ANNEX I.

3.1 RSSEs supporting developers in change tasks

Software developers, especially newcomers, often encounter difficulties in their change tasks. They usually have to understand the existing code, implementing the required modifications without breaking something in the process.

Thus, they need assistance to accomplish their first tasks. However, allocating an experienced member to assist newcomers could be expensive and not always possible for a long time period. And even for experienced members, locating the software artifacts relevant to the changing task at hand could be an error-prone and a time consuming task. In this perspective, RSSEs can be helpful by providing useful software artifacts. In the following, we describe some of the selected tools supporting these goals.

Mentor (Malheiros et al., 2012) assists newcomers in the realization of their first tasks by recommending solved change requests and their related source files. The process is triggered explicitly by developer's request. It starts with an open change request composed of different fields (summary, description and developers' comments) written in a natural language text. Those fields are concatenated and compared to a set of solved change requests stored in a database which are processed beforehand into an advanced statistical model. The tool analyzes every stored change request and version control files in order to identify and store the associated relations (i.e. by scanning the commit messages). To do so, a heuristic based on regular expressions is used. To identify change requests similar to the open one, a comparison is performed using an entropy measure. The entropy of the open change request is calculated using the advanced statistical model of every stored change request. The similar changes are then classified according to their entropy scores and presented to the developer. Clicking on a recommended change request, the tool shows the associated revisions and source files.

Hipikat (Cubranic et al., 2005) is a similar RSSE which assists newcomers by recommending artifacts from the current project. A project memory is formed implicitly by all the artifacts of the project under development and links between those artifacts (e.g. file revisions which implemented a particular change request). An artifact could be: (1) a change task artifact (e.g. feature request and bug report), (2) a source file version stored in Control Version System (CVS), (3) a message (e.g. emails and forums), or (4) an other document (e.g. design documents). The links between project artifacts are inferred by Hipikat using different heuristics such as:

- clustering (e.g. clustering change task requests that have been fixed within the same time window);
- regular expression based heuristics for instance used to match change requests with the related file versions, etc.

The project memory is built and updated automatically, and the recommendations can be generated as soon as any part of the memory is formed. The recommendation process is triggered explicitly by the developer, selecting an artifact from the project under development and sending a request from a contextual menu. The selected artifact is tokenized, converted into a Bag of Words (BoW) which is processed to form a weighted vector and projected into a semantic space using Latent Semantic Indexing (LSI) which is an advanced Natural Language Processing (NLP) technique. Artifacts similar to the selected one are identified for two main situations: (1) either the selected artifact already exists in the project memory and Hipikat recommends the related artifacts based on the links stored in the project memory, or (2) the text similarity of two weighted vectors is performed using a cosine similarity measure. The similar artifacts identified are classified and ordered according to the similarity confidence which could be numeric (i.e. text similarity score) or descriptive (e.g. "high-checked in within the last five minutes"). Artifacts are clustered according to their types (e.g. CVS files) and each presented artifact provides a reason (e.g. check in to bug resolution).

The other RSSEs in this category work more closely with source code in order to recommend useful source code artifacts or elements. For instance, **MI** (Lee and Kim, 2015) recommends files to edit by leveraging developer interaction histories (i.e. viewed and edited files). The goal of the proposed tool is to help developers even before editing a file. The key idea is that the recommender monitors the last viewed (and edited if exist) files in a sliding window (referred as viewed-edited-sized sliding window) and looks for previous tasks with similar viewed files in a corpus formed of interaction traces. These interactions are stored as a pair of sets (viewed files, edited files) associated to a given task (e.g. bug fixing tasks). Based on association rules, the tool recommends the files edited in the identified similar tasks.

Once the developer edits a file from those recommended, the edited file is then considered in the new context with the viewed files to recommend other files to edit. For instance, Figure 3.1 shows an example in which developers have performed three tasks T0, T1 and T2, and a developer is performing a task T3. As the current viewed files (d, b, c) have been viewed in the task T0, MI then recommends the files that were edited in T0 (c, e).

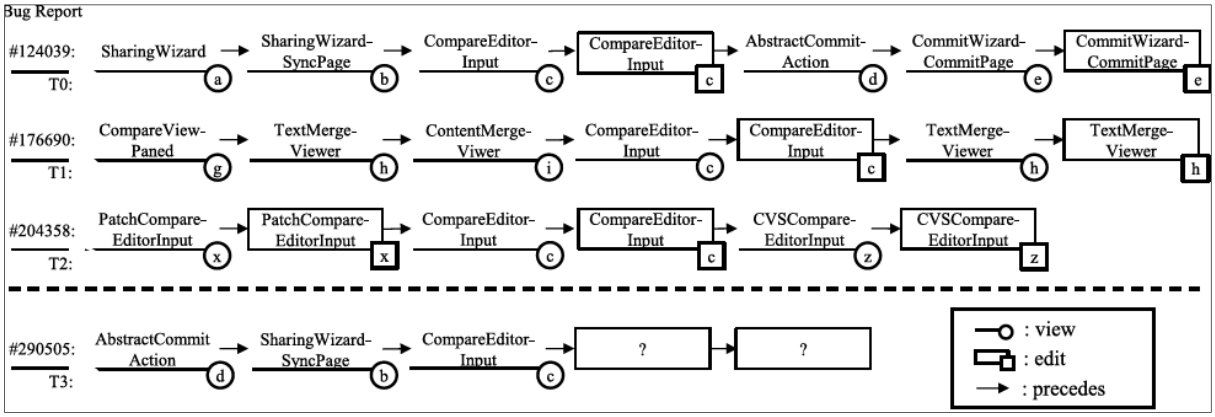

Figure 3.1 An example that shows the files programmers view and edit while performing tasks. This example is simplified from the actual interaction traces of bug reports #124039, #176690, #204358, and #290505 in the Eclipse Bugzilla system
Extracted from Lee and Kim (2015, p.316)

Mendel (Lozano et al., 2011) is a similar RSSE which assists developers in their change tasks. The tool detects what is missing in an entity (e.g. method, class) by analyzing related entities based on inheritance dependencies and identifying the most common properties. For instance, the family set of a class is composed of the direct superclass and its direct subclasses (i.e. siblings, nephews / nieces). In order to identify the common traits (dominant and recessive) in a family, Mendel sets two threshold measures. The identified traits (i.e. structural properties: types, naming or structural conventions) are thus recommended to be considered by the entity under development.

Other recommenders may help developers in planning complex modification tasks such as pragmatic reuse activities.

For instance, the recommender proposed in (Holmes et al., 2009) assists developers in planning their reuse tasks based on the structural relevance and the reuse cost of an element (e.g. class or method). The tool is an extension of **Suade** which is a topology analyzer of software dependencies, merged with Gilligan which is an environment for planning pragmatic reuse tasks. The proposed approach automatically recommends software elements that can be reused when a developer is triaging elements in the reuse plan. The tool extracts methods and fields of the triaged elements in Gilligan and analyzes the topology of the structural dependency graph built by Suade in order to identify their structural neighbors (e.g. callers of each method in the triaged elements). A degree of interest and a reuse cost measure are assigned to the identified neighbors based on their structural relevance (specificity and reinforcement) and the number of their descendants. The weighted elements are then ranked before being presented to the user.

3.2 RSSEs supporting developers in API usage

To improve their productivity, developers often tend to reuse existing libraries instead of writing the code from scratch. However, they may encounter difficulties in instantiating a particular object or in invoking related API methods which are usually complex to use and not well documented. In this perspective, RSSEs can be of great help to get familiar with API methods usage.

Altair (Long et al., 2009) recommends API methods based on structural information. The developer sends a query that contains an API method signature. The tool analyzes the query and extracts the data accessed by the given method that will be used to compute pair-wise overlap with API methods stored in a corpus. An API in the corpus is represented as a bipartite access graph where there are two types of vertices: functions and data, and edges represent access relations. Some heuristics are used to extend this graph into an augmented access graph (Figure 3.2 shows an example of augmented access graph). API methods identified by the overlap treatment are ranked and only the top ten results are selected. They are then clustered according to their purpose before being presented to the developer.

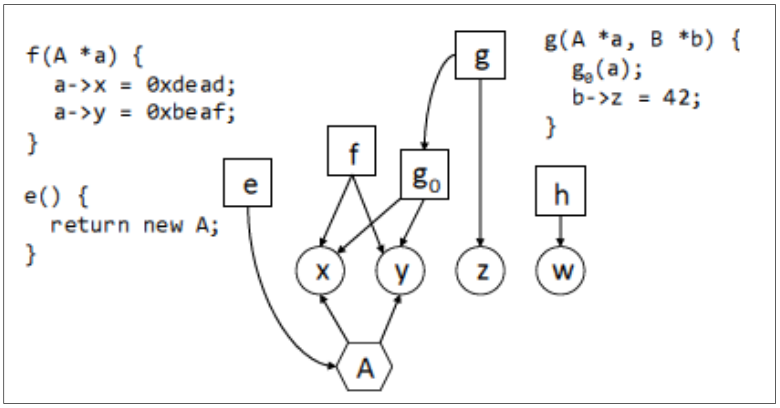

Figure 3.2 Augmented access graph. *e, f, g, g0, h* represent functions, *x, y, z, w* represent data, and *A* represents a composite type Extracted from Long et al.(2009, p.205)

Other RSSEs in this category assist developers in API usage by suggesting code snippets. For instance, **MAPO** (Zhong at al., 2009) recommends API usage patterns and their related code snippets. The developer selects an API method name from the body of the method under development and sends a request from the contextual menu. Methods and class names are extracted and sent as a query to a corpus containing API patterns. The corpus is formed of open source projects invoking API methods. A code analyzer is used to retrieve API method call sequences that will be clustered into patterns according to similarity of methods and classes' names computed by Levenshtein distance. The patterns and its associated sequences are presented to the developer as links which lead, when clicked, to the related code snippets.

Strathcona (Holmes et al., 2006) is another RSSE that recommends relevant API usage examples of source code according to the context of the task at hand. These examples are generated from a repository of existing applications that use the API. The developer selects a code snippet in the file under development and sends a request from the contextual menu. The tool extracts the structural context from the selected source code snippet. The structural context is a set of syntactic elements (e.g., the method signatures, the names of the types that declare those methods, etc.). Strathcona uses a set of heuristics based on structural facts similarity, e.g. a CALLS heuristic mines the repository to retrieve code snippets that make the same method calls as the structural context.

The repository of source code is indexed by the structural facts which are extracted in the same manner as the context. The similar code snippets identified are then ranked and only top ten results are presented to the developer as an example of usage with an illustrative graphical view and the rationale for selecting each example.

APISynth (Lv et al., 2014) assists developers in correctly instantiating API objects by providing a set of API Method Invoking Sequences (MISs). The developer sends a query that contains source and destination types and the tool returns a sequence of code statements that instantiate a new object of the second type starting from an object of the first type. To recommend these sequences, a repository of existing projects using the API is presented as a Direct Acyclic Graph (DAG); called also Weighted API Graph (WAG). DAG is a connected graph where nodes represent API methods and edges are built if the output type of an API method matches an input type of another API method (an example is shown in Figure 3.3). The tool uses the Key-Path based Loose algorithm to identify DAGs appropriate to the given query. The identified paths are ranked according to some criteria such as the path length, i.e. the shortest path represents the higher rank.

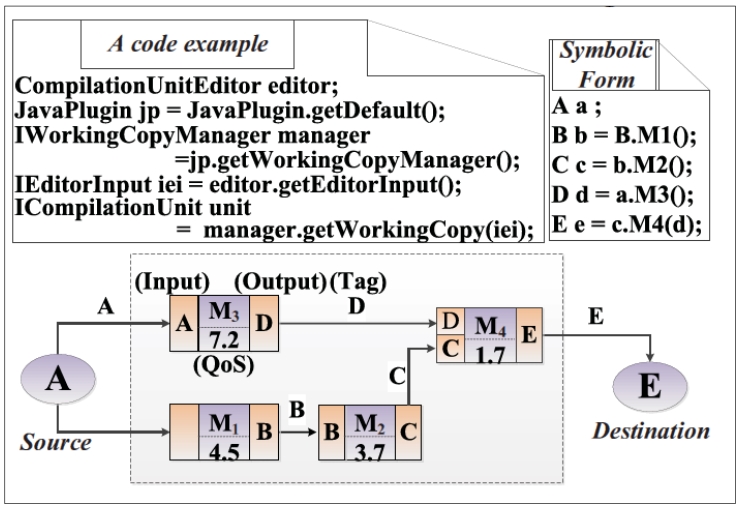

Figure 3.3 Example
Extracted from Lv et al. (2014, p.596)

3.3 RSSEs supporting developers in refactoring tasks

Software refactoring aims to maintain and understand software systems by restructuring the existing source code. However, selecting the appropriate refactoring operation that could be relevant to the current project is a challenging task due to the lack of documentation and large code bases. In this perspective, RSSEs can be helpful by providing relevant refactoring opportunities.

In (Bavota et al., 2014), the authors proposed an approach that recommends refactoring opportunities based on team development activity. The basic assumption of this approach is that code entities (e.g. methods or classes) modified by the same team could be extracted and grouped together in a separate module (e.g. class or package). A team is defined as a group of developers who has worked on the same source code entities. A code analyzer is used to parse the source code of a project and to extract its change history and the associated authors within a specified time window. The retrieved changes (e.g. methods added, removed or updated) are tokenized and rendered into a BoW. A clustering technique is used to group developers working on same code entities into teams. The output of this technique is a tree, called dendrogram, where the leafs represent developers and the remaining nodes are the possible clusters (i.e. teams). In order to recommend refactoring opportunities, a detection algorithm is used, for instance the algorithm detects methods edited by the same team, if the number of these methods is superior to a threshold then those methods can be extracted to form a separate class. Thus, the approach recommends code entities to be extracted (e.g. methods, classes).

Thies and Roth in (Thies and Roth, 2010) proposed another approach to support refactoring tasks by providing rename refactoring opportunities. Based on variable assignments, the key idea of this approach is that a variable assigned to another usually points to the same object and if both variables are declared with the same type, they are likely used for the same purpose. The recommender analyzes the source code of a project and extracts variable assignments' statements in order to build an assignment graph.

Using this graph, the tool detects the assignment of two variables having the same type but different names. In order to identify the variable name to recommend, some techniques are used such as misspelled names, synonyms names, etc.

DCLFix (Terra et al., 2012) is another recommender that assists developers and maintainers in refactoring tasks by recommending refactoring guidelines. The tool provides recommendations to remove violations detected by constraints defined using the Dependency Constraint Language (DCL). These constraints are checked using a companion tool DCLCheck. The recommendation process is triggered implicitly when an architectural violation is detected. DCLFix then extracts statements from the source code where the constraint has been violated. To generate recommendations, some preconditions have to be validated, for instance, to extract a method that will be moved to another class, the tool should find an appropriate class for the extracted method. To do so, the tool computes the similarity between the method and the class using the Jaccard similarity coefficient and returns the class with the highest value of similarity coefficient.

Other recommenders may help developers in repairing programs affected by evolution and refactoring modifications such as **SemDiff** (Dagenais and Robillard, 2008). This tool captures adaptive changes (e.g. method additions and deletions) and recommends similar adaptations (e.g. replacement methods). The proposed approach is based on the assumption that « ... calls to deleted methods will be replaced in the same change set by one or more calls to methods that provide a similar functionality. » (Dagenais and Robillard, 2008, p. 482). The developer selects a method call that no longer exists in the source code of the project or the framework and sends a request to a corpus containing changes' sets. The corpus is formed of log files which are preprocessed using clustering and mining heuristics to retrieve change sets. These files are clustered according to log entries that occur in the same given time window and share the same user and log message. To find a replacement for the method selected by the developer, the tool looks for all the methods where a call to the selected method was deleted and gathers all the added method calls in these methods.

The identified methods (i.e. added methods) are weighted, filtered by removing methods that have a weight value below a given threshold (set at 0.6) and then ranked before being presented to the developer as a list of changes (e.g. replacement methods). Clicking on a recommendation, the tool shows the source file where the recommended change replaced the old method call.

3.4 RSSEs supporting developers in solving exception failures, bugs, conflicts and testing tasks

During the software development process, developers often encounter problems in fixing bugs and solving exception failures that appear in their IDE. They usually look for solutions in resolved previous bugs or in Question / Answer (QA) web resources. Yet, the exception or bug context is not considered. In such cases, developers should enter the suitable query to get useful solutions which often leads to failed searches.

Some recommenders support developers in solving exception failures by providing relevant information in QA web resources. For instance, **SurfClipse** (Rahman and Roy, 2014) is a recommender that assists developers in solving exception failures by providing, in a first step, search queries, and in a second step, relevant web pages from QA web resources. The tool provides interactive and proactive working modes. When an exception occurs in the IDE, the tool analyzes the encountered exception and its context code to recommend a ranked list of search queries. SurfClipse extracts tokens (e.g. method name, class name) from the stack trace and builds a token graph. A weight is assigned for each token using the Degree Of Interest (DOI) and a variation of PageRank algorithm. Only the top scoring five tokens are selected to formulate a ranked list of search queries by combining each three of them. When the developer selects a search query from the recommended list, a search process is launched to collect results from three search engines (Google, Yahoo and Bing) and the QA website StackOverflow. A dynamic corpus is formed of the collected results which are analyzed and ranked according to their content relevance (i.e. title and textual content of each page). Only the top 30 results are presented to the developer as a list of web pages links. Clicking on a link, the tool shows the content of the web page.

Other recommenders assists developers in fixing their bugs, such as **AutoFix** (Pei et al., 2015). The recommender, integrated in the EiffelStudio development environment, automatically finds bugs and suggests source code patches. AutoFix recommends two types of changes: changes to the implementation and changes to the specification. First, to find bugs, the developer enters the name of the class to be analyzed in a typing box. Then, AutoFix calls a companion tool AutoTest to generate unit tests for the given class and to execute them. The faults identified by the failed executions are displayed to the user. To generate fixes for these faults, AutoFix identifies locations that are responsible for the bug using dynamic analysis of the executed tests and builds some fix suggestions that will be injected into the failing locations. The validation of these snippets is performed using the set of tests generated by AutoTest and the snippets that pass this validation are presented to the user.

Some recommenders help developers in bug triage tasks, such as **Sibyl** (Anvik and Murphy, 2011) which recommends developers to whom assign the report, affected components and subcomponents, and other project members who may be interested in the report. To do so, the recommender builds a corpus of bug reports collected from an issue tracking system. These reports are mined in order to extract the associated features and then clustered according to features categories. Using the title and the description, each report is tokenized, with the removal of all stop words, and converted into a weighted vector using TF-IDF technique. Given a new report (tokenized and weighted in the same manner as the corpus), the recommender uses a machine learning algorithm (Support Vector Machines) to retrieve similar reports. The reports are then ranked and the ones with similarity scores higher than a given threshold are selected to be recommended. The recommendations (i.e. developers, components and other project members) are then presented to the user as drop-down boxes.

Some approaches assist developers and maintainers in investigating and resolving conflicts when merging parallel source code versions, such as **ScoreRec** (Niu et al., 2012) which recommends a ranked list of conflicting software entities based on cost and benefit estimations.

To quantify the cost estimation of conflict resolution, the recommender uses the existing tool Semantic Diff which takes as an input two versions of a procedure (i.e. a method) and returns the semantic differences between them, i.e. dependence pairs where « ... a pair of variables, (*x*, *y*), forms a dependence pair if *x*'s value after execution of the procedure depends on *y*'s value before the procedure is executed. » (Niu et al., 2012). A procedure presents a conflict if its parallel versions return different sets of dependence pairs, and the cost estimation of fixing this conflicting procedure is computed based on the number of the identified inconsistent dependence pairs. Then, the benefit estimation of a procedure is determined according to change impacts caused by global variables. Finally, the identified set of conflicting procedures is ranked according to the benefit/cost ratio before being presented to the developer. Clicking on a particular procedure, the tool displays a detailed explanation in a separate window.

Other recommenders, such as **Test Tenderer** (Janjic and Atkinson, 2013) help developers in testing tasks by leveraging previously created test cases to proactively provide test case suggestions. The recommendation process starts when a user developed a class and starts writing tests, the tool extracts automatically all method invocations in the class under test (i.e. the developed class) that will be sent to SENTRE, an existing search engine for unit tests implemented previously by the same authors. The search engine run a query against a corpus formed of test cases which are analyzed in order to extract the associated interfaces. Based on the extracted contextual information, the tool search for interfaces semantically similar to the one under test (i.e. written by the developer) and identifies the associated test cases using the dependencies stored in the corpus. The tool then executes the developer's test against the classes under test associated to the test cases identified in the search, and the tests that pass this step are then ranked before being presented to the developer. The ranking is performed according to some criteria such as: the interface similarity, the overlap in the statement execution sequence between the test written by the developer and those returned by the search, etc.

3.5 RSSEs recommending reusable software components and components' design

While most existing RSSEs tend to recommend source code snippets or artifacts during the code phase, developers may need software components' design or implementation. Various mining tools and search engines can help them, yet they are context independent which often lead to an overwhelming quantity of results or no results at all.

In (Ichii et al., 2009), the authors propose an extension to the search engine SPARS-J in order to help developers in finding a component suitable to their needs, components related to this component and code examples to reuse these components. The tool extracts implicitly a developer's browsing history when the developer starts navigating through the results returned by the search engine SPARS-J. Recommendations to the developer are made using the collaborative filtering technique, based on the assumption that developers who have similar browsing histories (or navigation sessions) require similar components. To do so, the current navigation session is compared to a collection of developers' browsing histories which are stored as ratings in a corpus, i.e. a component is rated 1 if the developer browsed the source code of the component. The identified components are then ranked and filtered, i.e. the components that the developer has already seen are eliminated.

Rascal (McCarey et al., 2005) is another tool that provides reusable software components by tracking developers usage histories. When the developer edits a source file, the recommender extracts implicitly the invoked methods in it and forms a vector by counting the number of times each method has been invoked in the active class. The resultant vector is used as the relevant context to retrieve similar components from previous projects stored in a corpus which is continually updated as new classes or projects are developed. Those projects are processed in the same manner as the context in order to form a matrix (Figure 3.4 shows an example). Recommendations are made using a collaborative filtering technique based on the assumption that similar users, i.e. classes, tend to invoke same methods. These recommendations are then ranked using content-based filtering technique by examining the order in which each class has invoked the methods to recommend.

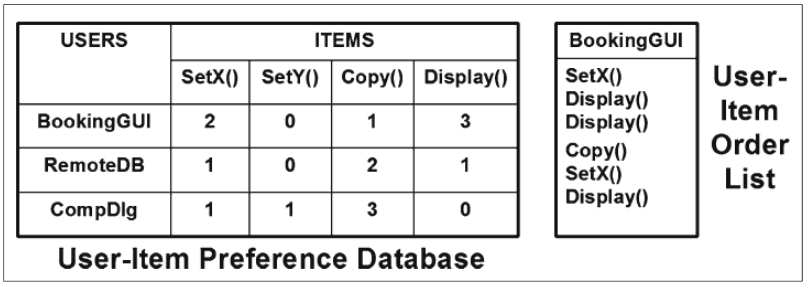

Figure 3.4 Sample user-item database
Extracted from McCarey et al. (2005, p.264)

A-SCORE (Shimada et al., 2009) is a similar recommender that leverages existing source code to provide reusable software components. When the developer edits a source code file, specifically when a comment or a statement delimiter is typed, the tool extracts implicitly a set of code elements (e.g. comments, field statements, method invocations, etc.). The extracted elements are tokenized to generate a bag of words, which is converted into a query and then into a weighted vector with the weight being the code element's distance from the cursor. The resultant vector is used to query a corpus formed of source code of existing projects which are parsed in the same manner as the contextual information (i.e. each source file is tokenized and converted into a weighted vector). A weighted matrix is built using the extracted code elements, where rows represent code elements (e.g. comments, field statements, method invocations, etc.) and columns are software components (i.e. classes) with the weight being the number of occurrences of every code element in a given class. This matrix is then projected into a semantic space using Latent Semantic Indexing (LSI). Recommendations are generated by computing cosine similarity of existing components to the contextual information, and then ranked according to their similarity scores before being presented to the developer.

Some other recommenders aim to propose components' design such as **Code Conjurer** (Hummel et al., 2010) which recommends software components' design by identifying the intersection of similar artifacts. When the developer edits a file, the tool extracts method signatures and class names that is sent as a query to the search engine Merobase.

In the same manner, Code Conjurer extracts method signatures of the search results and counts the number of occurrences of each method signature. Only method signatures that appear more often than a given threshold are selected. An explorer that contains components (i.e. classes) and its associated method signatures is displayed to the developer.

3.6 RSSEs assisting developers in exploring local codebases and visited source locations

Browsing web resources, documentations or searching in local codebases is time-consuming activity in software development. Various retrieval tools and search engines can help developers in doing so but again, they are context-independent. In this perspective, RSSEs help developers search efficiently by leveraging data from past and current browsing.

Sando (Ge et al., 2014) helps developers in exploring local codebases to find the relevant code snippets by recommending search queries. The proposed recommendation technique relies on the following data source components:

- Local dictionary of the codebase which contains terms that appears at least once in the codebase;
- Term co-occurrence matrix is formed of the terms collected from the codebase, each element in the matrix represents the count of two terms that appear in the same entity in the codebase; and
- Verb-direct-object pairs represent related verbs and objects, e.g. "open file", "create instance", etc.

The recommender supports pre-search and post-search recommendation modes. In the pre-search mode, when the developer issues a search query, the recommender retrieves the software entities whose indexed terms match with the given query and the recommended queries are listed in a drop-down menu. In the post-search mode, the recommender is triggered implicitly when the manual query entered by the developer fails.

In the same perspective, **Refoqus** (Haiduc et al., 2013) is a recommender that assists developers in reformulating queries. For a given query, the tool recommends a reformulation strategy to improve its performance. These strategies depend on the properties of the query (e.g. expansion strategy performed when the given query contains a single term). To provide recommendations, the tool requires a corpus formed of previous queries and their relevant results. For each query, Refoqus computes its property measures based on different weighting heuristics (e.g. Average Inverse Document Frequency, etc.), and then applies to each query four reformulation techniques: three of them support query expansion based on lexical similarity and weighting, and the last supports query reduction. The obtained results are compared to identify the best reformulation strategy for each query in the corpus. Hence, every query is defined by a set of property measures and a reformulation strategy. The obtained data is presented as a classification tree which is used to provide recommendations. When a new query is typed by the developer, Refoqus computes the property measures (in the same manner as the queries in the corpus) and retrieves the reformulation strategy to apply using the text retrieval engine Lucene.

Reverb (Sawadsky et al., 2013) is a recommender that provides web pages from the developer's previous browsing history that can be useful to the current development task. The tool extracts implicitly code elements that have been viewed by the developer and constructs an AST. Specific code elements are extracted from the AST (e.g., type declarations, method invocations, etc.) and used to form a query against a corpus that contains indexed web pages previously browsed by the developer. To generate recommendations, Reverb uses Apache Lucene similarity scoring that uses vector space model to match the formed query with the indexed web pages based on content similarity and the frequency and recency of page visits. The returned pages are ranked according to the visits of the developer (i.e., frequency and recency). The gathered results are ranked and then a links' list of the top ten scored web pages is presented to the developer.

NavClus (Lee et al., 2013) is a graphical code recommender which helps developers find unexplored source code locations that can be relevant to visit.

To do so, the tool considers the last sequence, without loops, of code elements (e.g. classes and methods) being viewed by the developer and uses it against a corpus of interaction traces. Those stored sequences are clustered using a k-nearest neighbor clustering algorithm. To retrieve clusters similar to the given query, the recommender uses a similarity metric based on TF-IDF. The identified code elements are then presented as a class diagram in a graphical view, clicking on a code element the tool shows the related source location.

Other recommenders help developers and maintainers in exploring and understanding a project, such as the recommender proposed in (Sora, 2015) which identifies and suggests the most important classes in a given project. The tool analyzes the project source code and represents it as graph where nodes are classes or interfaces and edges are static dependencies between them. To provide recommendations, the recommender ranks the graph nodes (i.e. classes) using the PageRank algorithm. The key idea is that a class which is used by many classes may represent a fundamental data and can be considered as an important class. Similarly, a class which is using other important classes can be considered as an important one. Finally, the tool displays only top 20 ranked classes.

3.7 Other RSSEs

3.7.1 RSSE assisting developers in software prototyping activities

Some recommenders aim to assist developers in software prototypes development. Prototyping activity usually contains two phases:

- identification of a candidate features set to implement a product; and
- implementing a selection of the identified features.

Some tools support developers only in the first phase, while others provide assistance in both phases, such as the approach proposed in (McMillan et al., 2012) which leverages open-source repositories to mine feature descriptions and its associated software components.

In the first phase, the developer sends a request for recommendations by describing the features of the new product. The written text is then tokenized, stemmed and rendered into a BoW that will be used to query a corpus. This corpus is formed of source code and specification documents retrieved from open-source repositories. Documents are mined in order to retrieve feature descriptions which are then clustered as many feature descriptions can represent similar functionality. The retrieved information is presented as a binary matrix where rows represent products and columns are features. Similarly, the source code is mined to retrieve modules (i.e. packages) associated to the mined features whose relations are presented as matrix where rows are modules and columns are features, and also dependencies between modules which are presented as a direct graph.

Using a cosine similarity, the tool generates a preliminary check list of features and recommends features with a score higher than a given threshold (fixed 0.6). The developers select the ones that seem relevant to the new product description. Based on the selected features, the recommender uses the content based filtering to generate additional recommendations. All the selected features are then used in the second phase to recommend the associated modules by requiring the modules/features matrix. To minimize coupling costs of recommended packages, the tool assigns weights (i.e. coupling cost values) to the vertices of the dependencies graph (i.e. direct graph) using a variation of the PageRank algorithm.

3.7.2 RSSE assisting developers in tagging software artifacts

In software engineering, tagging has been proven to be a useful mechanism in searching and classifying software artifacts, since it provides annotations to tag artifacts relevant to a given software activity. In this perspective, RSSE can be of great help for developers.

For instance, **TagRec** (Al-Kofahi et al., 2010) recommends tags not only for new work items but also untagged and tagged existing work items. A work item describes a development activity and contains a summary, a description of the activity, a tag and relevant software artifacts related to the given activity.

The tool parses every work item, tokenizes and stems its terms with removing grammatical terms and stop-words. The obtained BoW is used to build a correlation matrix where the correlation value between two terms is determined based on the number of work items in which these two terms occur together. To provide recommendations, TagRec uses the fuzzy set theory where « Each term defines a fuzzy set and each work item has a degree of membership in this set. The key idea is to associate a membership function for each work item with respect to a particular term. » (Al-Kofahi et al., 2010). The membership values range from 0 (i.e. no membership) to 1 (i.e. full membership) and are considered as the degrees of relevance of terms that are the most suitable to describe a given work item. The terms that have membership values higher than a chosen threshold are recommended to developers as tags.

3.7.3 RSSE recommending experts

In distributed software development, finding an expert of a given package or a piece of code is a challenging task due to the lack of knowledge sharing and synchronous communication in distributed teams which affect negatively the team's productivity. This problem can be addressed by RSSEs that identify and recommend people with the right knowledge.

For instance, **Conscius** (Moraes et al., 2010) helps developers in finding experts when they need assistance in a programming task by leveraging source code history, the project documentation and communication histories (mailing lists). To do so, the tool analyzes the content of the mailing lists and identifies the related source code and documentation (javadoc) using mining techniques. The obtained dependencies form a corpus that will be used when a developer writes a message to request recommendations. Conscius analyzes the typed message, identifies the referenced classes and finds classes and documentations (javadoc) related to the identified classes using the dependencies stored in the corpus. The tool then identifies a set of keywords in the javadoc and assigns a weight to each keyword based on its frequency in the document.

The obtained list of keywords is associated to top-level javadoc packages and is compared to the keywords extracted from the message written by the developer (i.e. context) using fuzzy similarity technique. The package with the highest similarity score is used to identify developers who sent emails with the requested knowledge (i.e. the package with the highest similarity score). Then, the tool assigns to each identified developer two main scores:

- communication score computed according to the number of messages sent by the developer that contain the identified package; and
- development score which depends on the number of commits on the classes extracted from the context in the CVS source files.

The tool recommends developers with highest score (sum of the two scores mentioned above) as experts.

3.8 Conclusion

In this chapter, we presented a sample of the analyzed RSSEs. We classified these RSSEs according to their goals into seven categories:

- Supporting developers in change tasks;
- Assisting developers in API methods usage;
- Supporting developers and maintainers in refactoring tasks;
- Solving exception failures, bugs and testing tasks;
- Recommending software components and components' design;
- Assisting developers in exploring local codebases and visited source locations; and
- Various goals such as software prototyping activities, tagging software artifacts and recommending experts.

CHAPTER 4

RESULTS ANALYSIS

This chapter presents various features for each key step in RSSE building. First, we describe the context extraction process by identifying different efficient ways used to retrieve useful information and converting it into data. Then, we present the recommendation engine that takes into account the extracted context to provide recommendations.

4.1 Context Extraction

4.1.1 What is context in RSSE ?

Context is a multifaceted concept which can be specified differently. Bazire and Brézillon in (Bazire and Brézillon, 2005) tried to understand this concept. However, they noticed that « ... it is difficult to find a relevant definition satisfying in any discipline. Is context a frame for a given object? Is it the set of elements that have any influence on the object? Is it possible to define context a priori or just state the effects a posteriori? Is it something static or dynamic? » (Bazire and Brézillon, 2005). In a previous work (Dourish, 2004), Dourish proposed some answers and stated that context concepts could be presented following two main axis: representational and interactional.

Representational concept describes the features of the environment surrounding a particular activity but it is separate from the activity itself. It is perceived as a form of information which can be known and predefined as it does not change significantly over short time period (e.g. codebases and repositories). Thus, it is considered delineable and stable as it does not vary from instance to instance of an activity or an event.

Interactional concept, unlike representational concept, is derived from the activity as it is actively produced and maintained through the task at hand.

Thus, it is perceived as a relational property linking objects or activities. It is defined dynamically and it is specific to each activity or action.

As we consider context extraction as the retrieval process of information relevant to a given programming activity, we consider the interactional axis as the most appropriate. We consider context the volatile data being edited (or even browsed) as it is more susceptible to convey useful information about the programming task at hand. In the following section, we propose and present some key components to define context extraction.

4.1.2 Context Extraction: An overview

In this section, we represent the identified techniques and concepts using feature modeling formalisms which are proposed in (Czarnecki et al., 2006). Feature modeling is an approach for describing and modeling requirements of products especially in software product lines development. A feature model is presented as a tree called feature diagram. Figure 4.1 shows the top-level feature diagram which presents an overview of different key components characterizing the context extraction process. The legend presented under the figure explains the notation. We consider that the context extraction process is triggered by an event that will initiate the collection of an input which can be treated to produce an output. The input has a scope which represents the highest level of hierarchy to reach in order to retrieve the relevant information (e.g. file, package, project), and may require the extraction of specific elements within the scope (e.g., method invocations within the file).

4.1.3 Trigger

The context extraction process can be triggered explicitly or implicitly by an event. The explicit, i.e. reactive mode is usually prompted by an action that calls for recommendations, e.g. clicking on a button "Query for Recommendations" from a contextual menu (e.g. Hipikat (Cubranic et al., 2005)). However, the implicit, i.e. proactive mode is activated by an event which is monitored by the RSSE and considered as an implicit call for recommendations, e.g., browsing a code element in Mendel (Lozano et al., 2011).

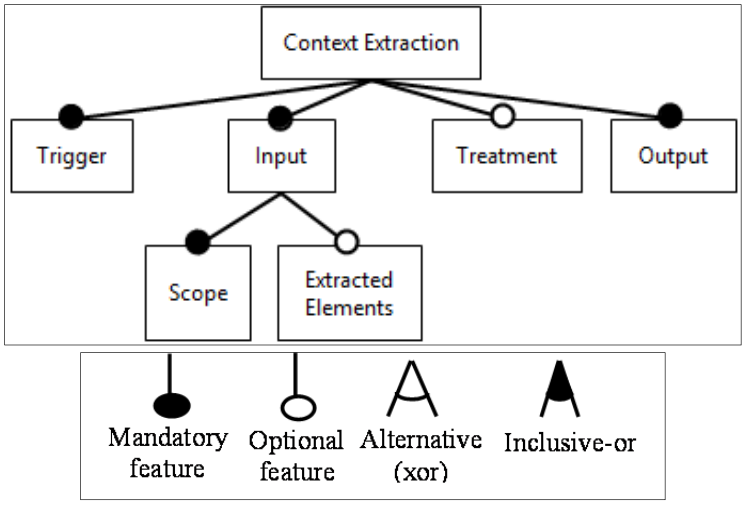

Figure 4.1 A Feature Model for context extraction in RSSEs
Extracted from Maki et al. (2015, p.154)

Figure 4.2 shows the trigger's feature diagram. In the reactive mode, the context extraction is triggered by an explicit command which is usually clicking on a button that can be in a contextual menu (e.g. Hipikat (Cubranic et al., 2005) and MAPO (Zhong at al., 2009)), a search box, i.e. sending a query (e.g. APISynth (Lv et al., 2014) and DebugAdvisor (Ashok et al., 2009)), or a custom view of the workspace (e.g. Mentor (Malheiros et al., 2012)).

In the proactive mode, a variety of triggering events has been noticed, it ranges from:

- browsing a web page or a code element (e.g. class or method) such as MI (Lee and Kim, 2015); to
- editing source code (e.g. Test Tenderer (Janjic and Atkinson, 2013)); to
- scrolling with a mouse (e.g. NavClus (Lee et al., 2013)); to
- typing a query in a search box (e.g. Sando (Ge et al., 2014)) or getting no results from a search; to
- run-time events (e.g. thrown exception in ExceptionTracer (Amintabar et al., 2015)).

Some RSSEs provide both modes, such as SurfClipse (Rahman and Roy, 2014) which is an Eclipse plugin with a default pro-active mode that is triggered when an exception is thrown.

This mode can be deactivated by the developer. It is then replaced with a reactive mode, which requires the selection of an exception from the console view.

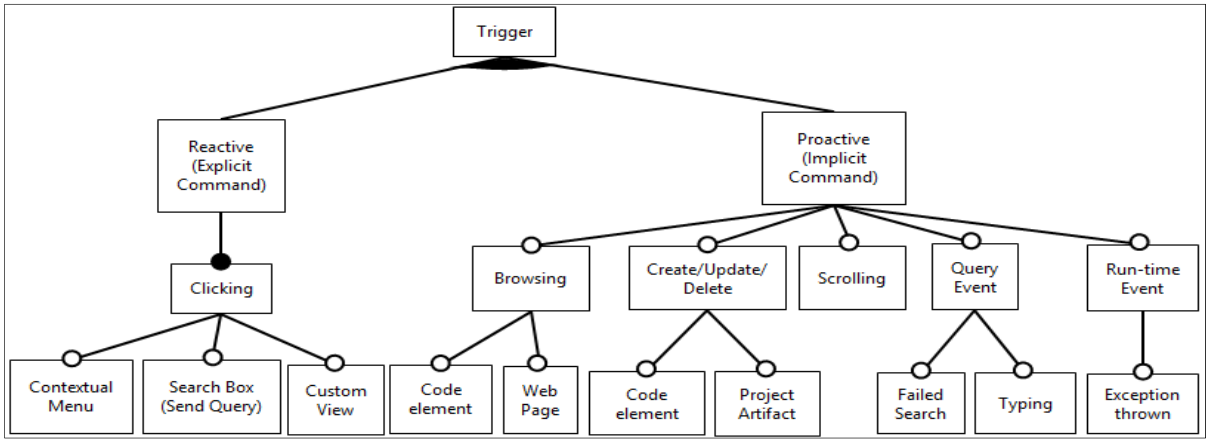

Figure 4.2 Features of the trigger
Extracted from Maki et al. (2015, p.156)

4.1.4 Context input

As we previously mentioned, a context input is retrieved within a given scope and possibly involves the extraction of some specific elements.

4.1.4.1 Input Scope

The scope represents the highest level that has to be considered in the context input to get relevant information. Figure 4.3 shows the main features of the scope which are related to space and time aspects. Space aspect can be:

- a code snippet (e.g. Strathcona (Holmes et al., 2006));
- code hierarchy that ranges from a line of code to the whole project (e.g. Suade (Holmes et al., 2009));
- project artifacts such as logs, bug reports (e.g. Hipikat (Cubranic et al., 2005)), change requests (e.g. Mentor (Malheiros et al., 2012)), etc.;

- elements in the workspace such as the stack trace (e.g. ExceptionTracer (Amintabar et al., 2015)) or a search box (e.g. Altair (Long et al., 2009)).

Regarding time aspect, the scope can be limited to a session (e.g. from the launch of the IDE in Reverb (Sawadsky et al., 2013)) or to a time interval (e.g. month, year, etc. (Bavota et al., 2014)).

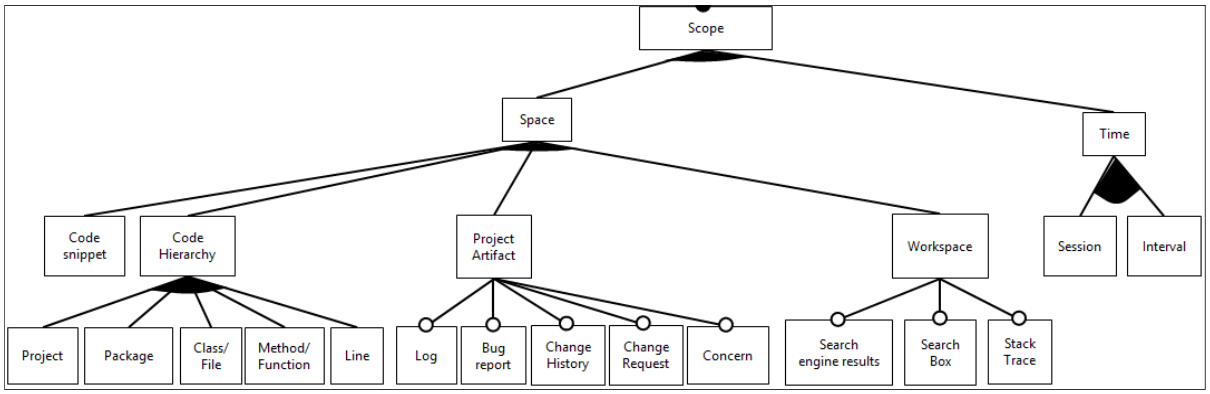

Figure 4.3 Features of the input scope
Extracted from Maki et al. (2015, p.156)

4.1.4.2 Specific Elements to extract

Sometimes, only a portion of the scope is considered relevant for extracting the context. Figure 4.4 shows the features of the specific element to extract. This element can be:

- a code element that ranges from identifiers (e.g. Concern-Detector (Robillard and Manggala, 2008)), to statements (e.g. DCLFix (Terra et al., 2012)), to methods signatures (e.g. SemDiff (Dagenais and Robillard, 2008)) to exceptions (e.g. (Cordeiro et al., 2012));
- a query that could be written in natural language text (e.g. Conscius (Moraes et al., 2010)) or in customized structure such as debugger output (e.g. DebugAdvisor (Ashok et al., 2009));
- a sequence of browsed elements (e.g. Reverb (Sawadsky et al., 2013)).

However, some RSSEs may extract a combination of various elements (e.g. Mendel (Lozano et al., 2011)).

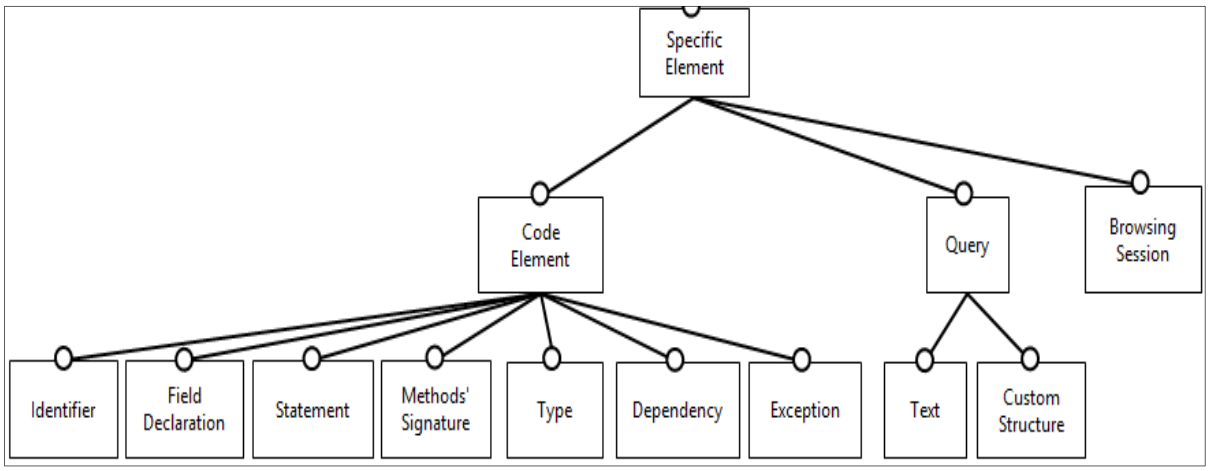

Figure 4.4 Features of the specific element to extract
Extracted from Maki et al. (2015, p.156)

4.1.5 Treatment

The treatment of the context input mainly involves parsing, weighting and filtering techniques as presented in figure 4.5.

Parsing is generally the first step which usually involves tokenization techniques. Tokenization can be performed on text such as project artifacts (e.g. (Denninger, 2012)) or source code elements such as identifiers (e.g. (Heinemann and Hummel, 2011)). For instance, the CamelCase convention is frequently used to split identifiers formed of several words into distinct words, i.e. an identifier *getMessage* may be tokenized into *get* and *message*.

More complex parsing techniques can be used to further extract specific code elements, to retrieve project data (e.g. change history of a project in a specific time window and the associated authors in (Bavota et al., 2014)) or custom features (e.g. DebugAdvisor (Ashok et al., 2009).

Weighting can be a second step that follows parsing. Weighting techniques include simple counts of code elements, e.g. number of times a method has been invoked in a given class (e.g. Rascal (McCarey et al., 2005)), or more complex techniques, e.g. probabilities assignment to terms in a set of documents using TF-IDF (Thompson and Murphy, 2014).

Filtering is usually used to remove or classify data. It may include a stemming technique which replaces related terms with a unique representative (e.g. *values* and *valued* can be presented as *value*) (De Souza et al., 2014), or clustering techniques (Bavota et al., 2014).

To those main techniques, we can add other techniques such as binary tagging which indicates the presence of an element (e.g. SPARS-J (Ichii et al., 2009)), or advanced NLP techniques such as LSI (e.g. SurfClipse (Rahman and Roy, 2014)).

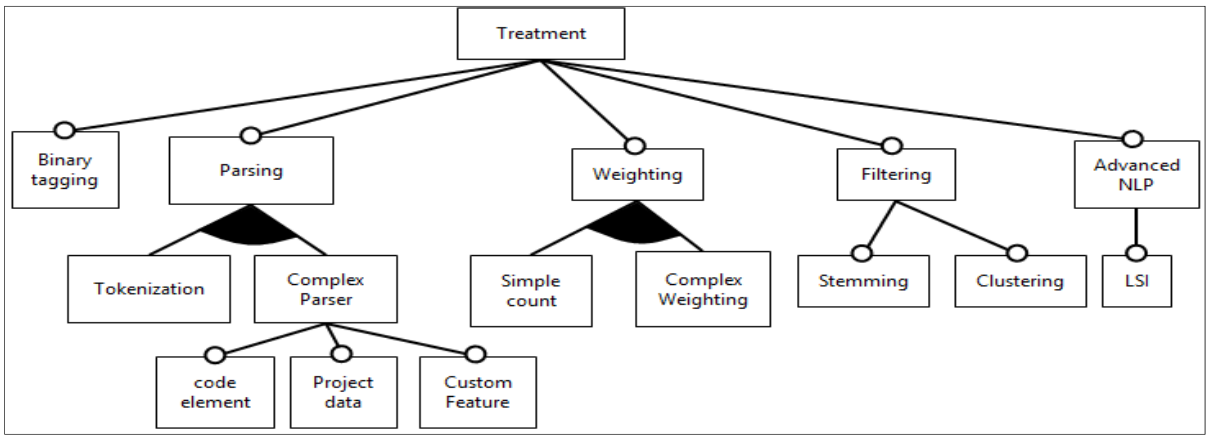

Figure 4.5 Features of the input treatment
Extracted from Maki et al. (2015, p.157)

4.1.6 Output

Figure 4.6 shows the different categories of data structures used as an output of the context extraction process (i.e. structures to which the context input is transformed). The main categories are sets, weighted vectors, sequences, queries, trees and graphs.

Sets reveal minimal treatments performed and usually are the collection of initial elements retrieved from the input scope (e.g. Mendel (Lozano et al., 2011), Suade (Holmes et al., 2009), Altair (Long et al., 2009), etc.). Binary vectors can be viewed as another type of sets, yet in such case, the complete alphabet is known and then 1 indicates the presence of an element (e.g. (Heinemann and Hummel, 2011)). Bag of words are usually the output of tokenization treatment where the input is split into a set of terms. They are usually used with simple input such as queries including natural language text (e.g. Conscius (Moraes et al., 2010)) or method names (e.g. MAPO (Zhong at al., 2009)), or selected code snippets.

Weighted vectors can be viewed as hash tables with keys being terms, code elements (e.g. method invocations) or other data and values being simple counts such as the number of times a method has been invoked in a class (e.g. Rascal (McCarey et al., 2005)) or the result of more complex operations (e.g. ImpRec (Borg, 2014)).

The structure of retrieved data can be more complex such as trees and graphs, for instance:

- an abstract syntactic tree (AST) (e.g. Reverb (Sawadsky et al., 2013));
- a dendrogram (Bavota et al., 2014) which is a tree where the leafs are specific elements (e.g. methods, developers, etc.) and the remaining nodes are possible clusters of those elements;
- an assignment graph where the nodes represent variables and the directed edges represent references (e.g. (Thies and Roth, 2010)); and
- a graph of tokens which represents a set of entities or terms and their relationships (e.g. SurfClipse (Rahman and Roy, 2014)).

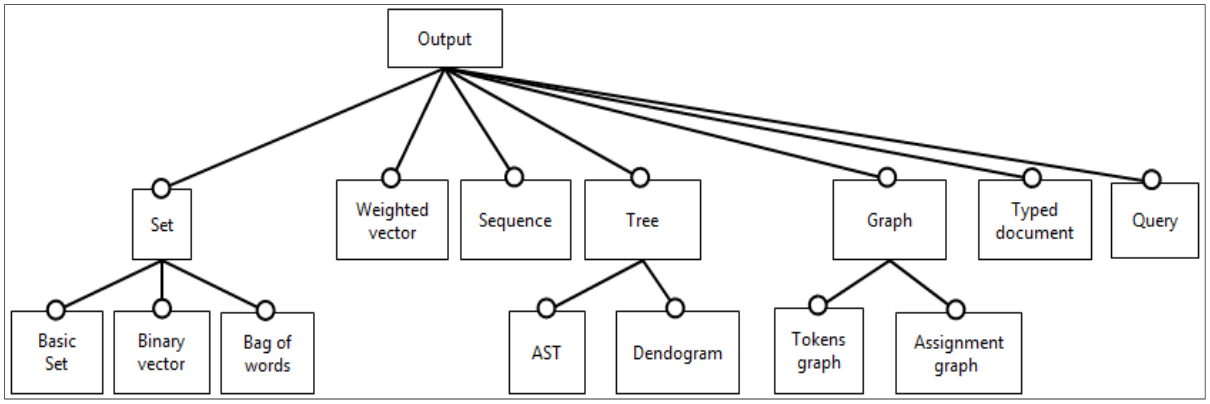

Figure 4.6 Features of the output
Extracted from Maki et al. (2015, p.156)

Based on the classification we proposed in this section, we built the table presented in ANNEX II, which summarizes the context extraction process in the tools we studied.

4.2 Recommendation Engine

4.2.1 Recommendation Engine: An overview

We consider that the recommendation engine contains a recommendation component that performs a treatment in order to generate recommendations which can be filtered or ranked before being presented to the developer. The recommendation engine possibly involves a corpus which is a set of raw data that can be treated in order to generate an output (processed data) used to recommend items. Figure 4.7 shows the top-level feature diagram which presents an overview of features characterizing the recommendation engine component.

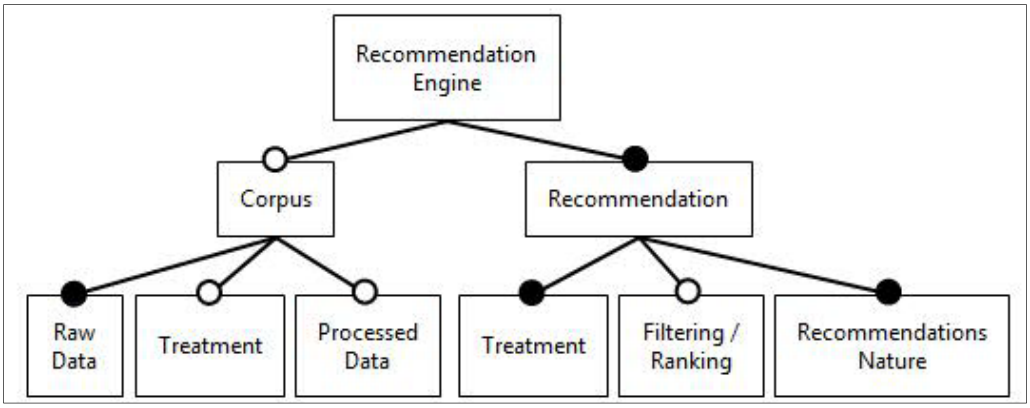

Figure 4.7 A Feature Model for the recommendation engine in RSSEs

4.2.2 Corpus

In (Mens and Lozano, 2014), the authors stated that the corpus of Source Code-based Recommendation System (SCoReS) « ... is program code, yet it is sometimes complemented with additional sources of information such as change management or defect tracking repositories, informal communications, local history, etc. » (Mens and Lozano, 2014). We consider a corpus every data source used to provide recommendations that may include project artifacts, external libraries, QA websites, etc. A corpus can be generated and updated automatically by leveraging project artifacts related to a given project such as Hipikat (Cubranic et al., 2005), or created manually by the developer, for instance, Concern-Detector (Robillard and Manggala, 2008) allows the developer to add code elements in a view related to a concern that is used later to generate recommendations. In other cases, a corpus can be dynamic when it consists of search results collected from a search engine, as those results depend on the given query.

As we previously mentioned, a corpus is a set of raw data that can be possibly treated and rendered as a processed data which will be used to generate recommendations. In the following subsections, we detail the corpus features.

4.2.2.1 Raw Data

Raw data is a set of primary information that will be used to provide recommendations. As shown by Figure 4.8, raw data may range from (1) source code of existing projects or external libraries (e.g. source code of an API in Altair (Long et al., 2009)), to (2) project artifacts that can be solved change requests (e.g. Mentor (Malheiros et al., 2012)), source file versions stored in CVS, bug reports (e.g. Hipikat (Cubranic et al., 2005)), concerns (e.g. Concern-Detector (Robillard and Manggala, 2008)), logs of debugger sessions, developers' messages (e.g. emails and forums) or other documents (e.g. design documents), to (3) developer interaction histories such as viewed or edited code elements (e.g. MI (Lee and Kim, 2015), NavClus (Lee et al., 2013)), to (4) unit test cases related to existing projects (e.g. Test Tenderer (Janjic and Atkinson, 2013)).

Some RSSEs tend to leverage information in web resources such as QA web sites (e.g. Stack Overflow). In such cases, raw data can be:

- questions and answers of a QA web site (e.g. ExceptionTracer (Amintabar et al., 2015));
- a set of collected results related to a search query (e.g. SurfClipse (Rahman and Roy, 2014), Code Conjurer (Hummel et al., 2010)); or
- browsing histories of visited web pages (e.g. Reverb (Sawadsky et al., 2013)).

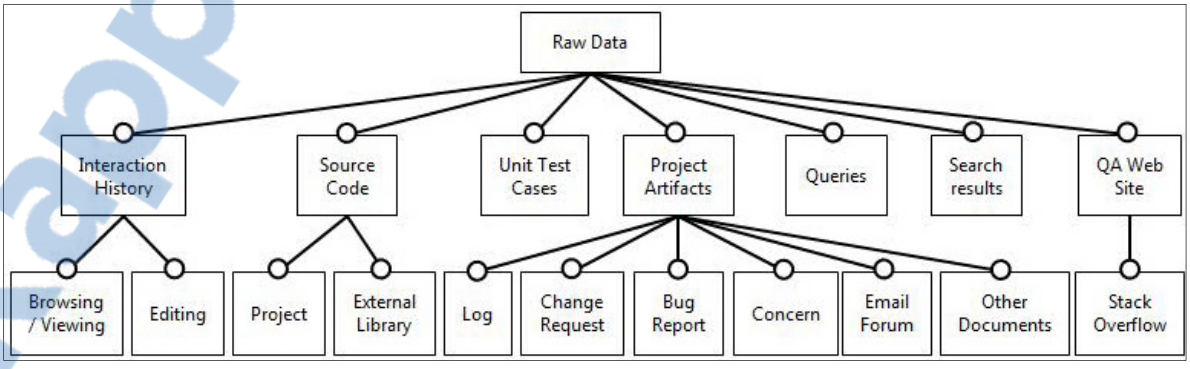

Figure 4.8 Features of the raw data

4.2.2.2 Treatment

The raw data can be processed using different techniques as presented in Figure 4.9. This treatment mainly involves the usage of some heuristics such as:

- mining of relevant information that will be compared to the context of the current development task, for instance mining API method invocations sequences (e.g. MAPO (Zhong at al., 2009)), mining questions and its related answers from QA web sites (e.g. (Cordeiro et al., 2012)), etc.;
- clustering such as clustering bugs fixed within the same time window (e.g. bugs fixed within the last six hours) or clustering similar artifacts based on structural similarity (e.g. Hipikat (Cubranic et al., 2005)); and
- regular expressions that can be used to identify links between change requests and the associated version control files (e.g. Mentor (Malheiros et al., 2012)).

Sometimes, the raw data is processed in the same manner as the context using the same techniques, such as:

- parsing which mainly involves tokenization of code elements such as identifiers (e.g. (Heinemann and Hummel, 2011)) or more complex parsers (e.g. DebugAdvisor (Ashok et al., 2009));
- stemming (e.g. (Denninger, 2012), (De Souza et al., 2014));
- weighting that may include simple counts of code elements such as number of times a method has been invoked in a given class (e.g. Rascal (McCarey et al., 2005)), or more complex weighting operations such as probabilities assignment to terms in a set of documents (e.g. Mentor (Malheiros et al., 2012)); and
- binary tagging which indicates the presence of an element (e.g. SPARS-J (Ichii et al., 2009), Javawock (Tsunoda et al., 2005)).

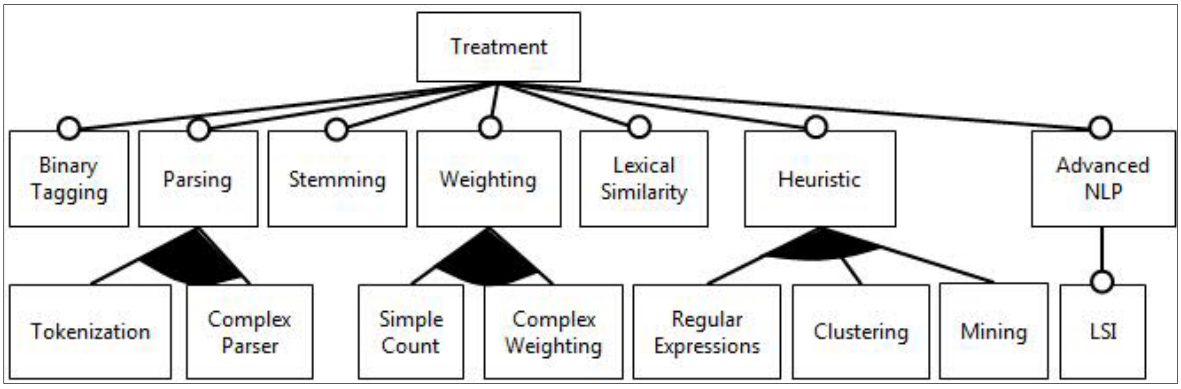

Figure 4.9 Features of the raw data treatment

4.2.2.3 Processed Data

Figure 4.10 shows the different categories of the processed data, i.e., data produced by the treatment of the raw data in the corpus. The main features identified are weighted vectors and matrices, patterns (or clusters), trees and graphs.

Weighted vectors usually present document artifacts by assigning weights to terms (or symbols) in each document in order to indicate its importance in the document and possibly in the entire collection of documents (e.g. Hipikat (Cubranic et al., 2005), Sibyl (Anvik and Murphy, 2011), etc.).

Matrices can be viewed as hash tables relating two different types of keys that can be code elements (e.g. row being class names and columns being method invocations) or other data. A matrix can be: (1) binary where each row consists of a binary vector, for instance "1" indicates that the method has been invoked in the class of a given row (e.g. Javawock (Tsunoda et al., 2005)), or (2) weighted where values can be simple counts, for instance the number of times a method has been invoked in the class of a given row (e.g. Rascal (McCarey et al., 2005)).

Patterns are usually the output of clustering treatment, for instance clustering method calls sequences (e.g. MAPO (Zhong at al., 2009)) or similar browsing histories (e.g. NavClus (Lee et al., 2013)).

The data stored in the corpus can be presented as graphs or trees, for instance:

- an augmented access graph where vertices are functions and data and edges represent access relations (e.g. Altair (Long et al., 2009));
- a Direct Acyclic Graph (DAG) which is a connected graph where nodes represent API methods and edges are built if the output type of an API method matches an input type of another API method (e.g. APISynth (Lv et al., 2014));
- a relationship graph which links description bugs with the associated source files, functions and authors (e.g. DebugAdvisor (Ashok et al., 2009)); or
- a binary search tree relating terms extracted from a codebase (e.g. Sando (Ge et al., 2014)).

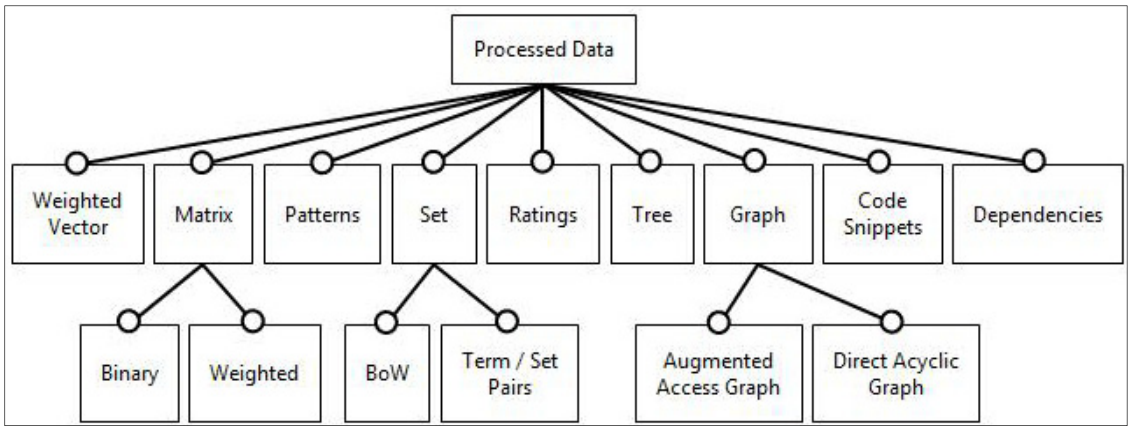

Figure 4.10 Features of the processed data

4.2.3 Recommendation

As we previously mentioned, the recommendation engine includes a recommendation component that performs a treatment in order to generate recommendations which can be filtered or ranked before being presented to the developer.

4.2.3.1 Treatment

In order to generate recommendations, we distinguish two types of treatments: (1) treatments based on inference, and (2) no-inference based techniques. Figure 4.11 shows the different features of recommendation treatment.

Inference-based techniques are usually based on the comparison of the extracted context of the current development task with the process data stored in the corpus. These techniques mainly involve:

- lexical similarity which computes the distance between two terms using simple scoring functions such as Levenshtein distance (e.g. MAPO (Zhong at al., 2009)), or more advanced measures such as entropy measure (e.g. Mentor (Malheiros et al., 2012));
- structural similarity which usually represents the textual similarity between two vectors and can be computed by simple measures such as the Hamming distance (e.g. (Heinemann and Hummel, 2011)), or more advanced measures such as the scoring function of Apache Lucene (e.g. Refoqus (Haiduc et al., 2013)); and
- weighting techniques which are often used to indicate importance of entities (or tokens) in a graph or results returned by a search engine, for instance TF-IDF (e.g. NavClus (Lee et al., 2013)), PageRank algorithm (e.g. SurfClipse (Rahman and Roy, 2014)), etc.

To these techniques, we can add traditional techniques like collaborative filtering (e.g. Javawock (Tsunoda et al., 2005)), content-based filtering (e.g. (McMillan et al., 2012)) and search algorithms such as Key-Path based Loose algorithm (e.g. APISynth (Lv et al., 2014)), or advanced overlap ranking techniques (e.g. Altair (Long et al., 2009)).

No inference-based techniques mainly involve simple operations to generate recommendations such as: (1) association rules technique (e.g. MI (Lee and Kim, 2015)), (2) binary tagging (e.g. AutoFix (Pei et al., 2015)), (3) defining a threshold (e.g. Mendel (Lozano et al., 2011)), or (4) performing a simple count (number of occurrences) to select relevant information that will be recommended (e.g. Code Conjurer (Hummel et al., 2010)).

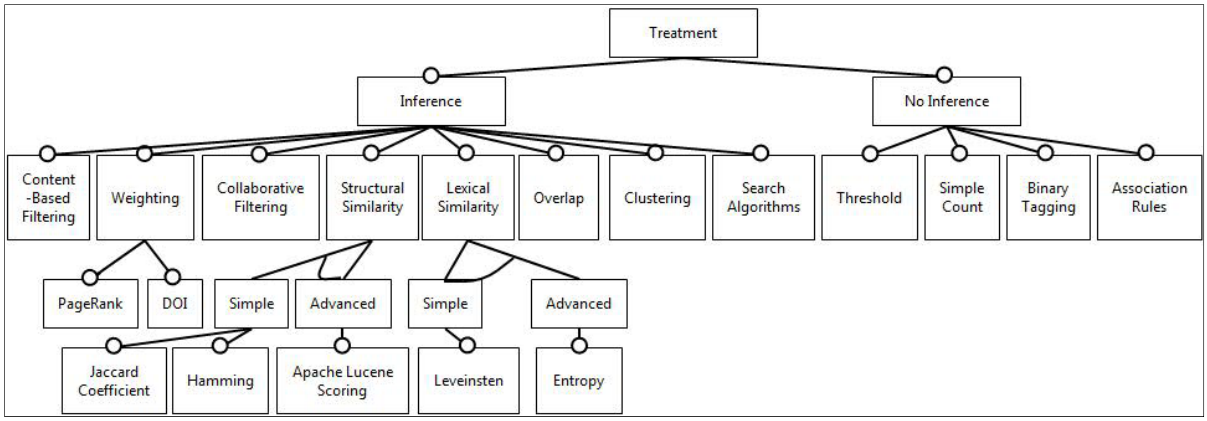

Figure 4.11 Features of the recommendation treatment

4.2.3.2 Filtering / Ranking

The set of relevant information selected by the recommendation treatment can be ranked and / or filtered before being presented to the user. Figure 4.12 shows the different features of the ranking and filtering processes.

Ranking usually reveals the classification of the data returned by the recommendation treatment component. This classification can be performed according to:

- the score computed in the treatment phase (e.g. A-SCORE (Shimada et al., 2009), Sibyl (Anvik and Murphy, 2011), Selene (Murakami et al., 2014), etc.);
- number of occurrences (e.g. (Heinemann and Hummel, 2011));
- path length for instance the shortest path in a given graph represents the higher rank (e.g. APISynth (Lv et al., 2014));
- custom criteria such as coverage criteria in testing tasks (e.g. Test Tenderer (Janjic and Atkinson, 2013)); or
- a traditional recommendation technique like content-based filtering (e.g. Rascal (McCarey et al., 2005)).

Filtering involves the removal of some data returned by the recommendation treatment component.

It may be performed by defining a threshold such as selecting the top ten results (e.g. Hipikat (Cubranic et al., 2005)) or by eliminating components or results that have been already viewed or browsed by the developer (e.g. SPARS-J (Ichii et al., 2009)).

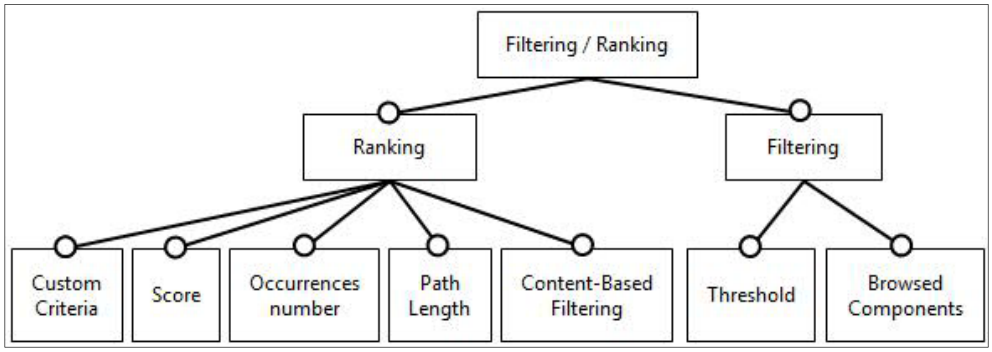

Figure 4.12 Features of the filtering and ranking

4.2.3.3 Recommendations Nature

Recommendations can be presented to the developer in various manners. Figure 4.13 shows the main features of recommendations nature that we have identified. Basically recommendations are presented as:

- a list of links that range from API methods (e.g. APISynth (Lv et al., 2014)), to code snippets (e.g. Strathcona (Holmes et al., 2006)), to project artifacts such as change requests (e.g. Mentor (Malheiros et al., 2012)), bug reports (e.g. DebugAdvisor (Ashok et al., 2009)), CVS files, and web pages that could be relevant to visit (e.g. (Cordeiro et al., 2012));
- clusters of API methods invoking sequences (e.g. Altair (Long et al., 2009)) or of project artifacts (e.g. Hipikat (Cubranic et al., 2005));
- an explorer including API patterns and its associated method invocation sequences (e.g. MAPO (Zhong at al., 2009)), software components and their associated method signatures (e.g. Code Conjurer (Hummel et al., 2010)), or concerns that could be relevant to the current development task (e.g. Concern-Detector (Robillard and Manggala, 2008)); or

- drop-down menus that contain a set of recommended search queries (e.g. Sando (Ge et al., 2014)).

Recommendations may be accompanied with additional information that can help the developer to examine whether a recommendation is relevant to the current task, such as:

- the rationale which usually explains the reason behind recommending a given item such as check-in close to a bug resolution (e.g. Hipikat (Cubranic et al., 2005)); and
- graphical views often presented as class diagram (e.g. Strathcona (Holmes et al., 2006), Navclus (Lee et al., 2013)).

We perceived that recommendations' presentation can be hierarchical, for instance recommendations presented as a list of change requests, when the developer clicks on a link of the list, the tool shows the CVS files related to the selected change request (e.g. Mentor (Malheiros et al., 2012)).

Based on the classification we proposed in this section, we built the table presented in ANNEX III, which summarizes the results of our analysis of the studied RSSEs according to our recommendation engine feature models.

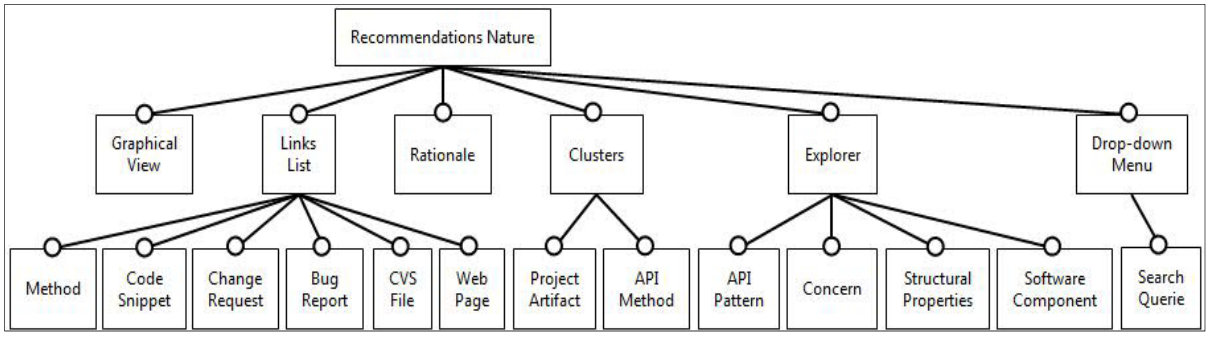

Figure 4.13 Features of the recommendations nature

CHAPTER 5

DISCUSSION

In this chapter, we discuss the results presented in the last chapter according to our research questions and we conclude with validity threats.

5.1 Results Synthesis

As we previously mentioned, the main goal of this study is to answer the following research questions:

- (Q1) : Which features characterize the context extraction process adopted by RSSEs?
- (Q2) : Which features characterize the recommendation engine used by RSSEs to provide recommendations?

We answer these questions in the following subsections.

5.1.1 Context extraction process

To answer the question (Q1), the different features, characterizing the context extraction process, and collected from our analysis of RSSEs sample are summarized in ANNEX II. We discuss below these results according to the high-level features of the context extraction process and the goal categories of the studied RSSEs.

Regarding the **trigger** which is the first feature characterizing context extraction process, the first observation is that it was difficult to recognize what it was for many tools like (Heinemann and Hummel, 2011) and (Thies and Roth, 2010). This aspect usually denotes that these tools are still in the proof-of-concept phase or they do not really capture a specific context but gather information at a project level in order to infer some generic recommendations.

Considering the sample of tools we analyzed, we noticed that slightly more tools are reactive in their support to developers. When we do see goal categories of RSSEs, most API usage tools work on a reactive mode except the work in (Heinemann and Hummel, 2011) which is unclear on its trigger. However, code exploration and software component recommendation tools all work on a proactive mode.

Regarding the **scope**, more than half of the tools use some levels of code hierarchy or code snippet as their scope, with file and project levels being the most commonly used (e.g. Code Conjurer (Hummel et al., 2010), Mendel (Lozano et al., 2011)). Search-related views and various project artifacts make for most of the rest of the tools. However, we noticed that only one tool restricts itself to a package to extract the relevant information. These findings reveal that the considered contextual information have evolved a bit since the review of (Happel and Maalej, 2008) and that many tools do now go beyond the file level. When we consider our goal categories, we perceive that change tasks tools tend to use larger scopes such as project-level artifacts or code hierarchy (e.g. Hipikat (Cubranic et al., 2005), Mendel (Lozano et al., 2011)), as they may need to make sure that the considered changes comply with unwritten rules within the project. Most of the tools recommending software components tend to consider the class under development as a scope as it may be easier in identifying similar classes. However, we noticed that the temporal dimension seems relevant in only three cases, two of which are related to refactoring ((Bavota et al., 2014), (Thies and Roth, 2010)).

Elements to be extracted are closely related to the scope but there are still some relevant differences. For instance, the number of tools that extract code source elements is far higher than the number of those that use source code to define their scope. All the analyzed tools extract specific code elements, with methods signatures (e.g. Altair (Long et al., 2009), Mendel (Lozano et al., 2011)), type information (e.g. APISynth (Lv et al., 2014), MAPO (Zhong at al., 2009)) and specific statements (e.g. SurfClipse (Rahman and Roy, 2014), Reverb (Sawadsky et al., 2013), etc.) being the most required elements. In many cases, these source code elements are extracted from queries, artifacts, etc.

As for the **treatment** used to process the extracted information, we noticed that it is very diverse ranging from common techniques like tokenization (e.g. Hipikat (Cubranic et al., 2005), MAPO (Zhong at al., 2009), etc.) to indexing and retrieval operations like LSI (e.g. SurfClipse (Rahman and Roy, 2014)). However, some relatively simple techniques, such as stemming, rarely appear. There was no clear observation related to the goal categories.

Regarding the **output**, sets, in particular bags of words, are the most dominant output. Weighted vectors usually go with some complex model (e.g. Hipikat (Cubranic et al., 2005)) and complex structures such as trees like (Heinemann and Hummel, 2011) and (Thies and Roth, 2010), and graphs (e.g. SurfClipse (Rahman and Roy, 2014)) are not as rare as could be thought. When we consider our goal categories, we perceive that API usage tools tend to use simple structures like sets and bags of words (e.g. Altair (Long et al., 2009), Strathcona (Holmes et al., 2006)) as it may be an easier way to match context terms with API terms. The same observation holds for tools recommending software components, which is expected given the similarity of purposes between these two categories. In contrast, RSSEs assisting developers in refactoring tasks tend to use trees and graphs, which is unsurprising as the refactoring operations to be recommended should be compliant with the complex structure of the code. On the other end of the spectrum, order and hierarchy seem to be relevant for tools assisting code exploration (e.g. Reverb (Sawadsky et al., 2013), NavClus (Lee et al., 2013)), as they tend to go for graphs, trees or sequences.

5.1.2 Recommendation engine

To answer the question (Q2), the different features, characterizing the recommendation engine component, and collected from our analysis of RSSEs sample are summarized in ANNEX III. We discuss below these results according to high-level features of the recommendation engine component and the goal categories of the analyzed RSSEs.

The first observation is that ten tools (21%) don't have a **corpus** (six of them are refactoring and debugging tools) to generate their recommendations.

In this case, corpus is blended with the context scope, as refactoring and debugging tools try to infer some generic recommendation based on the contextual information collected at a project level.

When we look at the **raw data** used to provide recommendations, we notice that more than half of the tools use source code or project artifacts (e.g. Hipikat (Cubranic et al., 2005), Altair (Long et al., 2009), MAPO (Zhong at al., 2009), etc.). Interaction histories, results returned by a search engine and QA web sites make for most of the rest of the tools. Regarding our goal categories, we perceived that change tasks tools tend to leverage project artifacts (e.g. Hipikat (Cubranic et al., 2005)); instead API usage tools use source code of the API or projects using a given API in order to provide recommendations.

In many cases, this raw data is processed in a high number of tools with source code and project artifacts being the most commonly treated (e.g. MAPO (Zhong at al., 2009), Rascal (McCarey et al., 2005)). In contrast, a dynamic corpus formed by results returned by a search engine is not processed.

As for the **raw data treatment**, we noticed that the most common techniques used are weighting (e.g. APISynth (Lv et al., 2014), Mentor (Malheiros et al., 2012)) and heuristics including mining, clustering and regular expressions-based (e.g. MAPO (Zhong at al., 2009), NavClus (Lee et al., 2013)). In some cases, RSSEs may use several techniques according to the type of the raw data (e.g. Hipikat (Cubranic et al., 2005)), and sometimes the same techniques used to treat the context (e.g. DebugAdvisor (Ashok et al., 2009)). There was no clear observation related to the goal categories.

Regarding the **processed data**, matrices, vectors, graphs and trees are the most dominant output of raw data treatment. As for our goal categories, we perceived that change tasks tools tend to use vectors (e.g. Mentor (Malheiros et al., 2012), Hipikat (Cubranic et al., 2005)) as it may be easier for matching contextual information with project artifacts.

Most of the tools recommending software components represent their processed data as a matrix (e.g. Rascal (McCarey et al., 2005), Javawock (Tsunoda et al., 2005), A-SCORE (Shimada et al., 2009)).

When we consider the **recommendation treatment**, the first observation is that the number of tools using inference-based techniques is far higher than the number using no inference techniques. Regarding the inference-based techniques, more than half of the tools use lexical or structural similarities to match contextual information with the process data stored in the corpus. Overlap (e.g. Concern-Detector (Robillard and Manggala, 2008), Altair (Long et al., 2009)), weighting (e.g. SurfClipse (Rahman and Roy, 2014), NavClus (Lee et al., 2013)) and traditional techniques like collaborative filtering (e.g. SPARS-J (Ichii et al., 2009), Rascal (McCarey et al., 2005)) and content-based filtering techniques (McMillan et al., 2012) make for most of the rest of the tools. Regarding our goal categories, we noticed that tools recommending software components tend to use collaborative filtering technique based on the assumption that developers who have similar usage or browsing histories require similar components (e.g. SPARS-J (Ichii et al., 2009), Rascal (McCarey et al., 2005)). Finally, RSSEs supporting change tasks and API usage tend to go with lexical and structural similarity techniques to provide recommendations.

Regarding the **ranking and filtering** of recommendations, the first observation is that the number of tools that rank and / or filter their recommendations before being presented to the developer is far higher than the number of those that do not. Also we noticed that most tools either rank (e.g. APISynth (Lv et al., 2014), DebugAdvisor (Ashok et al., 2009)) or rank and filter (e.g. Strathcona (Holmes et al., 2006), SurfClipse (Rahman and Roy, 2014), Rascal (McCarey et al., 2005), etc.) the recommendations set. As for the ranking, more than half of the tools rank their recommendations according to the similarity score computed by the recommendation technique. On the other end of the spectrum, threshold filtering is the most technique used as it may be easier to select, for instance, the top ten scored results. As for our goal categories, we noticed that most of the refactoring tools do not rank neither filter their recommendations as they may try to infer some generic recommendations.

Finally, when we consider **recommendations presented to the developer**, we notice that most tools present their recommendations in a hierarchical manner with lists of links and explorers being the most commonly used (e.g. Mentor (Malheiros et al., 2012), Hipikat (Cubranic et al., 2005), Concern-Detector (Robillard and Manggala, 2008), etc.), as it may be easier for the developer to quickly check the returned results and the related files by clicking on links. However, the rationale (e.g. Hipikat (Cubranic et al., 2005), Strathcona (Holmes et al., 2006)) and graphical views (e.g. Strathcona (Holmes et al., 2006), NavClus (Lee et al., 2013)) are rarely displayed. In some cases, tools are unclear on the presentation of the recommended results, and this again indicates that these tools are still in the proof-of-concept phase. Regarding our goal categories, tools supporting change tasks, API usage, debugging tasks and recommending software components tend to present their recommendations as lists of links or explorers so the developer can check the associated project artifacts (e.g. Hipikat (Cubranic et al., 2005)), or the related web page (e.g. SurfClipse (Rahman and Roy, 2014)), etc.

When we consider all characteristics, we notice the degree of variability ranging from the trigger to recommendations. There are no two tools that share significant similarities along that path, which means that (i) the analyzed RSSEs has succeeded in gathering a diverse set of recommenders or that (ii) RSSE researchers have not converged to some best practices in building RSSEs.

5.2 Validity threats

In this section, we discuss the factors having an impact on the validity of this study. We highlight some points that we consider as threats.

5.2.1 External validity

External validity threats are related to the possibility of generalizing the results of the experiment regardless of the study conditions.

In the case of systematic literature reviews, the external validity depends on the selected set of papers. By the choice of our exclusion criteria, we excluded papers that did not demonstrate a recommendation approach / tool. This tactic mitigate the external validity.

5.2.2 Internal validity

Internal validity threats are related to how well a study or an experiment is done. In the case of systematic literature reviews, internal validity refers to how well the results represent the true opinion expressed in literature. In this systematic review, it was not always easy to get the needed information from the published papers. As shown in tables summarizing the results (ANNEX II, ANNEX III), many papers are unclear about some important aspects and in some cases, there is no clear distinctions between the context and the corpus used to provide recommendations. Sometimes, there are no real context as researchers simply try to leverage information at a project level.

CONCLUSION

In this thesis, we conducted a systematic literature review to identify features characterizing each component we need to build an RSSE. To do so, we adopted the approach proposed in (Kitchenham, 2004) for conducting software engineering systematic literature reviews. Then, we classified the identified publications into seven categories according to the development task they support. We deeply analyzed each component starting from the data considered as the context to how it can be possibly treated and rendered as an output for the recommendation engine component. The latter is composed of three subcomponents: corpus which contains data used to get recommendations that possibly can be treated, recommendation technique which matches context with the data stored in the corpus, and possibly a filtering subcomponent that ranks and filters recommendations before being presented to the developer. This analysis led us to propose feature models that identify various design and implementation choices for each component.

To the best of our knowledge, it is to date the largest study on this topic. This work can be extended by including other important categories of recommendation tools such as code completion tools and by analyzing the evaluation techniques used to assess the recommender's quality.

In the mid-term, we plan to evaluate the different choices identified through experiments with the studied RSSEs. This study can be interesting to outline the best practices to design and implement an RSSE.

In the long-term, we can use the identified features to design and implement an RSSE framework which is a toolbox of reusable components for building RSSEs.

ANNEX I

RSSE DESCRIPTION

RSSE	Category	Description
(Yamada and Hazeyama, 2013)	Change Task	The tool supports developers and maintainers in understanding a software project by using the program package name (i.e. java package) to recommend relevant software artifacts such as exchanged messages and design documents. The developer selects a program package that s/he wants to understand and then selects a mode. The tool provides two modes: • a mode that retrieves documents and messages from source code (program package name): the recommender extracts candidate method signatures that are similar to the package name from CVS source code; • a mode that retrieves messages from documents: the tool extracts "artifact words" which are words that compose a communication message or a document, that are similar to the package name (in the same manner as the first mode). Then, the recommender retrieves the artifacts related to the extracted information using cosine similarity between vector space models of artifacts and the retrieved information. The recommendations are presented as a list of artifacts. Clicking on an artifact, the tool shows the content of the associated document or communication message.
(Thompson and Murphy, 2014)	Change Task	This approach helps developers when they start a new task by recommending one resource as the initial starting point (e.g. source file). This approach is based on the assumption that similar tasks, i.e. similar task descriptions, consider and change similar resources. To provide recommendations, a corpus is formed using previous tasks descriptions composed of different fields (title, description and developers' comments) written in a natural language text. These tasks are weighted using TF-IDF and rendered into weighted vectors. The tool mines the context associated to each task (in the previous tasks). A task context is defined as the set of resources selected or edited during the work on the given task. To generate recommendation, the recommender computes the similarity between a new task, processed in the same manner and rendered into a weighted vector, with previous tasks and ranks the results. Considering the top two ranked results, the tool computes the overlap of all their resources and returns one random resource.

RSSE	Category	Description
SDiC (Antunes et al., 2012)	Change Task	This RSSE recommends source code artifacts relevant to the task at hand by retrieving a context model presenting structural (artifact) and lexical (artifact terms) ontology of a selected artifact in the workspace. The relevance to the current task is captured through the analysis of the developer interactions with the artifacts in the workspace (e.g., opening or closing a file). Artifacts' similarity to the current context model is inferred from the structural similarity (distance between artifacts) and the lexical similarity (distance between terms). The identified similar artifacts are ranked according to a weighted sum of the similarity scores.
(Denninger, 2012)	Change Task	The proposed approach helps developers in finding code elements (methods and classes) relevant for a given change request (CR) using a combination of multiple predictors with machine learning. The recommender leverages information stored in issue tracking and version control systems. A corpus is formed of source code which is parsed in order to extract identifiers and comments (using CamelCase convention and stemming), CRs, source file revisions and links between them which are retrieved using mining algorithms. The proposed recommendation approach is based on the following three prediction approaches: • similarity of a given CR to former revisions based on textual similarity between the fields of CRs (title and description) and commit messages; • textual similarity of the given CR to former CRs; • textual similarity between the CR and both identifiers and comments extracted from source code. The scored code artifacts identified by each approach are then weighted to make them comparable using machine learning, and then only top scored n code artifacts are presented to the developer.
Concern-Detector (Robillard and Manggala, 2008)	Change Task	The recommender supports developers in their change tasks by recommending concern-related code elements (fields and methods) that overlap with code elements being currently modified. A concern is defined as a high level concept such as requirements and design decisions. A companion tool Concern-mapper is used to map existing source code elements to a concern. The mapping is performed manually by the developer who creates a view corresponding to a concern and adds any code element to the concern view.

RSSE	Category	Description
Change-Commander (Gall et al., 2009)	Change Task	The proposed recommender recommends method invocation changes when the developer inserts a method invocation in the file under development by leveraging change history. This recommender relies on two companion tools: (1) Evolizer which mines software archives (e.g. CVS, bug-tracking system, etc.), and (2) ChangeDistiller which extracts changes for each revision from the mined change history. Using these two tools, a corpus is formed of change type patterns which are clusters of changes, particularly those related to bug fixing, that frequently appear together. « For instance, when an if-statement with a certain condition is often put around a method invocation, the corresponding changes form a pattern. » (Gall et al., 2009). To provide recommendations, the tool looks for changes related to the method invocation entered by the developer and assigns a frequency (i.e. occurrence number) to each identified change. The list of method invocations is then ranked before being presented to the developer.
ImpRec (Borg, 2014)	Change Task	The recommender assists developers in Change Impact Analysis (CIA) when a change is required for an issue report. The proposed approach forms a corpus, similar to the project memory of Hipikat, presented as a network of software artifacts and trace links of previous CIA reports. The content of each artifact is indexed using Apache Lucene and weighted according to the importance of the artifact in the corpus. Given a new issue report, the tool identifies similar issue reports using the structural similarity of Lucene. Starting from the identified set of similar issue reports, ImpRec uses a breadth-first search algorithm to identify change impact candidates. These candidates are then ranked using some network measures and textual similarity.
(Heinemann and Hummel, 2011)	API Use	The proposed RSSE recommends API methods using information incorporated in identifiers. It is based on the assumption that code snippets using similar identifiers usually use similar methods. To do so, the tool parses the source code files of existing projects invoking API methods, and converts it into an Abstract Syntax Tree (AST). Traversing the AST, the tool analyzes method calls within every method body and, for every method call, it extracts all identifiers from the beginning of the method body to the line of the method call. In case of compound identifiers, the camel case convention is applied; identifiers parts composed of a single character are removed and the remaining parts are stemmed. The resulting BoW is used to form a matrix (columns represent identifiers terms and lines represent method calls).

RSSE	Category	Description
		Each row consists of a binary vector where "1" indicates that the identifier term occurs in the lines preceding the method call. This matrix is used to recommend API methods invocations. The recommendation process is triggered implicitly when the developer is editing a file. The tool extracts the context identifiers and forms a binary vector (same process described above). The hamming distance is used to compute similarity of two binary vectors. The methods invocations of the identified similar binary vectors are recommended and ranked according to their number of occurrences.
Selene (Murakami et al., 2014)	API Use	This recommender goes beyond the file under development to extract information relevant to the current editing activity. The tool includes source code related to the current method or class (e.g. the callers and callees of the current method). First, the tool monitors the source file under development, extracts code elements (e.g. method, file, class) and assigns to each retrieved element a degree-of-interest (DOI) which depends on the developer's activity (e.g. a selected or edited method has higher relevance score than others in the same file). These elements are then rendered into a weighted vector used to query a repository formed of open-source projects using a given API. The files of these projects are tokenized and converted into weighted vectors using the TF-IDF technique. Based on a cosine similarity, Selene identifies files similar to the query, and then retrieves lines that are similar to those in the file under development using a simple algorithm (i.e. the tool splits every result file into segments of 20 lines and computes similarity between the code under development and every segment). The identified code snippets are then displayed to the developer.
(Cordeiro et al., 2012)	Debugging	The tool assists developers when their code fails with an exception by recommending relevant information gathered from the QA web site Stack Overflow. The recommender is triggered implicitly when an exception stack trace appears in the IDE. The exception is analyzed and a bi-dimensional context is retrieved which contains: (1) Structural context that considers all the references identified in the exception stack trace; and (2) Lexical context which is composed of the names of said references. The extracted context is used to form a query for retrieving relevant answers stored in a corpus. This corpus is built from questions and answers extracted from the QA web site Stack Overflow which are usually composed of alternate blocks of text and source code. These blocks are analyzed and processed in order to identify exception stack traces.

RSSE	Category	Description
		If no stack traces were detected, the source code blocks are parsed to form the appropriate AST. The mined information (exception stack traces and source code) is indexed by the terms that represent the associated source code references. These terms are tokenized using the CamelCase convention, for instance a code snippet associated to the method reference Database.connect() is indexed by the terms database and connect. The formed corpus is queried when the context is extracted and only the search results containing the name of the exception are mined (a maximum of 200 results). These results are ranked according to a weighted sum of retrieval score (scoring function of Apache Lucene), structural and lexical scores. The final recommendations are presented as a list of relevant Stack Overflow web pages, clicking on a link, the recommender shows its content.
Exception-Tracer (Amintabar et al., 2015)	Debugging	The recommender helps developers in solving exceptions by automatically providing solutions. However, this tool does not only leverage information from the QA website StackOverflow but also open-source repositories (SourceForge). When an exception occurs, the tool analyzes the stack trace and extracts statements that caused the exception and then constructs a directed graph that represents objects used in the code. This graph is used to identify other objects involved in the exception failures, the type of each object and the invoked methods. Using the extracted information (i.e. the exception and contextual information in the code), the tool formulates two queries: • the first one is written in a specific language to mine source files and constructs for each file a directed graph that will be used to extract paths with limited sizes; and • the second query is written in a natural language text to search discussions in StackOverflow. The retrieved source files and web pages are then presented to the developer as list of links, clicking on a link, the tool displays the associated code snippet / discussion in StackOverflow.
(De Souza et al., 2014)	Debugging	The approach helps developers in solving their problems by providing a ranked list of questions / answers (QA) pairs retrieved from the QA website StackOverflow with respect to the developer's query. A given query, written in natural language text, is tokenized, stemmed and rendered into a BoW that will be used as a search query against a corpus. This corpus is built by retrieving QA pairs of the website StackOverflow. The content of each pair (title, question and answer, except the code snippets) is tokenized and stemmed.

RSSE	Category	Description
		As for the code snippets in questions and answers, the tool extracts the names of methods, classes and interfaces that will be tokenized using CamelCase and added to the resultant QA pair document. The search engine Apache Lucene is used to retrieve QA pair documents textually similar to the query by computing the Lucene's score. To improve the quality of the retrieved pairs, the tool considers the votes of the StackOverflow community. A final score (i.e. the arithmetic mean of Lucene's score and StackOverflow score) is assigned to each pair. Then, a ranked list containing the top 10 QA pair documents is displayed.
Debug-Advisor (Ashok et al., 2009)	Debugging	The recommender supports developers in fixing their bugs by recommending relevant information (e.g. people, source files, methods, etc.). The developer sends a query that could include kilobytes of unstructured data (e.g. natural language text) and structured data (e.g. debugger output). The recommender extracts from the given query a set of features using a feature parser. These features are formalized as typed documents that have the following four type structures: unordered bag of terms, ordered list of terms, weighted terms and key-value pairs. These typed documents are subsequently converted into bags of words which are used for similarity purposes with previously fixed bugs. A corpus is formed of bug reports and VCS source files. The bug reports are processed in the same manner as the query and indexed based on TF-IDF. The VCS is mined in order to retrieve version control revisions that were made to fix the bugs and to build a relationship graph which relates elements in the bug description (e.g. source files, functions, people, etc.). In the first phase, the recommender uses customized lexical similarity measures to identify fixed bugs that are similar to the given query. A variation of PageRank algorithm is performed on the relationship graph in order to assign weights to the graph entities. A ranked list of those entities, i.e. bug reports, source files, functions and people is recommended to the developer.
(Kpodjedo et al., 2008)	Debugging	The approach identifies the critical classes developers should focus on in testing tasks. Given two class diagrams of a system at different evolution levels, the tool builds a mapping by identifying classes that have been modified, added or deleted. Using this mapping, the tool assigns the following weights for each class, based on the assumption that frequently changed classes are fault-prone, (1) PageRank which measures the importance of a given class in the system, and, (2) Evolution Cost which evaluates class changes in a time period. The recommendations are presented as a graphical view (scatter-plot) to indicate the distribution of the identified classes.

RSSE	Category	Description
(Erfani et al., 2013)	Debugging	The proposed approach helps developers in testing tasks by providing unit test case examples. The developer selects a function to test and sends a request from the contextual menu. The tool extracts the selected method and applies to it any code clone treatment (e.g. weighting, tokenization, stemming, etc.) to get it ready for comparison with other functions stored in a corpus. A corpus is formed of existing source code projects and their associated unit test cases. Clone detection techniques are performed using an existing tool which is based on a detection threshold to identify cloned fragments. For each clone class, the tool identifies: (1) the cloned methods which have at least one unit test case, and (2) the clone fragments which have no unit test case. For these clone fragments, unit test cases are then recommended based on existing unit test cases associated to the clone class.
Javawock (Tsunoda et al., 2005)	Component	The recommender provides software components, in particular java components, based on collaborative filtering technique. However, this tool considers java programs (class files written by the developer) as users, java library class files used in the program as items, and uses both CF algorithms: user-based and item-based (unlike Rascal which uses only user-based technique). User-based method determines users with ratings (i.e. preferences) similar to those of the target user, predicts ratings of new items using ratings of similar users and then recommends new items that seem to be preferred by the target user. Item-based method determines items with ratings similar other items rated by the target user and predicts ratings of new items using ratings of similar items (already identified). Given a java program (uploaded by the developer), Javawock extracts the set of library class files' names used in the given program and uses it to query a corpus formed of existing java programs using java library classes. These programs are presented as a binary matrix where rows are programs and columns are library class files. Each value in this matrix is set to 1 if the given program uses the library class file in the associated column, otherwise it is set to 0. Using the collaborative filtering technique, the tool identifies similar java library class names and ranks them according to their similarity scores before being presented to the user as a list of links.

ANNEX II

RSSE ANALYSIS: CONTEXT EXTRACTION PROCESS

RSSE	**Catego-ry**	**Trigger** *CUD=Crea-te/Update /Delete*	**Scope** *S=Space W=Work-space PA=Project Artifact*	**Element** *C=Code Element*	**Treatment** *P=Parsing W=Weigh-ting F=Filtering*	**Output** *S=Set*
Mentor	Change Task	R/Custom View	S/PA/Chan-ge Request	Change Request Fields	W/Complex/ PPM	W. Vector
Hipikat	Change Task	R/C. Menu	S/Project Artifact		P/Tokenize W/Complex	S/BoW W. Vector
Yamada and Haze-yama	Change Task	R/Custom View	S/CH/Packa-ge	C/Package name		Set
Thomp-son and Murphy	Change Task	Unclear	S/Project Artifact	Task description Fields	W/Complex	W. Vector
Dennin-ger	Change Task	R.	S/PA/Chan-ge Request	Change Request Fields	P/Tokenize	S/BoW
MI	Change Task	P/Browsing	S/PA/Chan-ge History Time			Set, Query

RSSE	**Catego-ry**	**Trigger** *CUD=Crea-te/Update /Delete*	**Scope** *S=Space W=Work-space PA=Project Artifact*	**Element** *C=Code Element*	**Treatment** *P=Parsing W=Weigh-ting F=Filtering*	**Output** *S=Set*
Change-Com-mander	Change Task	P/CUD	S/CH/Class	C/Signature		Set
Mendel	Change Task	P/Browsing	S/CH/Pro-ject	C/Statement Signature, Dependency Type		Set
Concern Detector	Change Task	P/CUD	S/PA/Con-cern S/CH/Class	C/Identifier C/Signature		Set
ImpRec	Change Task	Unclear	S/PA/Issue Report		P/Tokenize W/Complex	S/BoW W. Vector
Suade	Change Task	P/Browsing	CH/Project	C/Signature, Field		Set
Altair	API Use	R/Search Box	S/W/Search Box	C/Statement C/ Signature	P/CP/Code Element	Set
MAPO	API Use	R/Contex-tual Menu	S/CH/Class	C/Identifier C/Signature, C/Type	P/Tokenize	S/BoW
Strath-cona	API Use	R/Contex-tual Menu	S/Code Snippet	Code		Set

RSSE	**Catego-ry**	**Trigger** *CUD=Crea-te/Update /Delete*	**Scope** *S=Space W=Work-space PA=Project Artifact*	**Element** *C=Code Element*	**Treatment** *P=Parsing W=Weigh-ting F=Filtering*	**Output** *S=Set*
Heine-mann & Hummel	API Use	Unclear	CH/Method	C/Identifier	P/Tokenize F/Stemming	B. Vector S/BoW
Selene	API Use	R/Custom view	S/CH/Class	C/Signature, Field	P/CP/Code Element W/Complex	W. Vector
API-Synth	API Use	R/Search Box	S/W/Search Box	C/Type	N/A	Sequen-ce
Bavota et al.	Refacto-ring	R.	S/PA/Chan-ge History Time		P/Token, Project Data F/Clustering	S/BoW G/Tree
Thies and Roth	Refacto-ring	Unclear	CH/Project Time	C/Statement		G/Tree
DCLFix	Refacto-ring	P/Run-time/ DCL Constraint Violated	CH/Project	C/Statement		Set
SemDiff	Refacto-ring	R/ C. Menu	S/CH/Class	C/Signature		Set
Cordei-ro et al.	Debug-ging	P/Run-time	S/W/Stack Trace	C/Exception	P/Tokenize, Code Elt W/Count, VSM	Query

RSSE	**Catego-ry**	**Trigger** *CUD=Crea-te/Update /Delete*	**Scope** *S=Space W=Work-space PA=Project Artifact*	**Element** *C=Code Element*	**Treatment** *P=Parsing W=Weigh-ting F=Filtering*	**Output** *S=Set*
Surf-Clipse	Debug-ging	R/Custom View P/Run-time	S/W/Stack Trace S/CH/Line	C/Exception C/Statement	P/CP/Code Element LSI	Graph
Except-ion Tracer	Debug-ging	P/Run-time	S/W/Stack Trace S/CH/Line	C/Exception C/Statement	P/CP/Code Element	Graph, Query
De Souza et al.	Debug-ging	R/Search Box	S/W/Search Box	Query	P/Tokenize F/Stemming	S/BoW
Debug-Advisor	Debug-ging	R/Search Box	S/W/Search Box S/PA/Logs	Query	P/Tokenize P/Custom Feature	S/BoW
AutoFix	Debug-ging	R/Box	S/W/Search Box	Query (class name)		Unclear
Sibyl	Debug-ging	Unclear	S/PA/Bug Report	C/Fields	P/Tokenize W/Complex	W. Vector
Score-Rec	Debug-ging	R.	CH/Project	C/Field	P/Complex	Set (pairs)
Erfani et al.	Debug-ging	R/Contextua l Menu	CH/Method	C/Statement	"any code clone technique"	"any clone output"
TestTen-derer	Debug-ging	P/CUD	S/CH/Class	C/Signature		Set
RSSE	**Catego-**	**Trigger**	**Scope**	**Element**	**Treatment**	**Output**

	ry	*CUD=Crea-te/Update /Delete*	*S=Space W=Work-space PA=Project Artifact*	*C=Code Element*	*P=Parsing W=Weigh-ting F=Filtering*	*S=Set*
Kpod-jedo et al.	Debug-ging	Unclear	S/Project artifact (Class diagrams)			Graph
SPARS-J	Compo-nent	P/Browsing	S/W/Search Results		Binary Tagging	Set
Rascal	Compo-nent	P/CUD	S/CH/Class	C/Signature	W/Simple Count	W. Vector
Java-wock	Compo-nent	R/Custom view	S/CH/Class	C/Class names	P/Complex	S/B. Vector
A-SCORE	Compo-nent	P/CUD	S/CH/Class	C/Signature, Field, Comments, Statement	P/Tokenizatio n W/Simple Count	S/BoW W. Vector
CodeCo-njurer	Compo-nent	P/CUD	S/CH/Class	C/Signature C/Type		Set
Sando	Explora-tion	P/Query	S/W/Search Box	Query		Unclear
Refoqus	Explora-tion	Unclear	S/W/Search Box	Query	W/Complex	W. Vector
Reverb	Explora-tion	P/Browsing	CH/Project Time	Code	F/Clustering	Sequen-ce
NavClus	Explora-tion	P/Scrolling	S/Code Snippet	C/Statement	P/CP/Code Element	G/Tree Query
RSSE	**Catego-**	**Trigger**	**Scope**	**Element**	**Treatment**	**Output**

	ry	*CUD=Crea-te/Update /Delete*	*S=Space W=Work-space PA=Project Artifact*	*C=Code Element*	*P=Parsing W=Weigh-ting F=Filtering*	*S=Set*
Sora	Explora-tion	R.	S/CH/Projec t	C/Dependen-cy	P/ CP/Code Element	Graph
McMilla n et al.	Prototy-ping	R.	S/W/Search Box	Query/Text	P/Tokenize F/Stemming	S/BoW
TagRec	Tagging	Unclear	S/Project Artifact	Work items Fields	P/Tokenize F/Stemming	S/BoW Matrix
Cons-cius	Experts	R.	S/W/Search Box	Query/Text	P/Tokenize	S/BoW

ANNEX III

RSSE ANALYSIS: RECOMMENDATION ENGINE

RSSE	**Category**	**Corpus**			**Recommendation Engine**		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
Mentor	Change Task	PA/ Change request	W/Complex/ PPM H/Regular expressions	W. Vector	I/LS/A/Entropy	R/Score	LL/Change Request
Hipikat	Change Task	PA/Log, Bug Report, Email Forum, Other Documents	P/Tokenize W/Complex H/Clustering H/Regular expressions	W. Vector	I/SS/Advanced	R/Score	C/Project Artifacts + LL Rationale
Yamada & Hazeyama	Change Task	SC/Project PA/Log	W/Complex	W. Vector	I/LS + SS	R/Score	LL/Project Artifacts

RSSE	Category	Corpus			Recommendation Engine		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
Thompson and Murphy	Change Task	Project Artifact	W/Complex H/Mining	W. Vector	I/SS/Simple, Overlap		One Project Artifact (unclear presentation)
Denninger	Change Task	SC/Project, PA/Change request,Log	P/Tokenize +Stemming H/Mining	Dependencies	I/SS/Lucene	R/Score F/Threshold	Project artifacts (unclear)
MI	Change Task	IH/Edited, Viewed	H/Mining	S/Set pairs	NI/Association Rules	R/Occurrences number	LL/Files
ChangeCo-mmander	Change Task	PA/Log, Bug reports	H/Mining, Clustering	Patterns	I/Weighting	R/Score	LL/Methods
Mendel	Change Task				NI/Threshold		LL/Structural Properties
Concern-Detector	Change Task	PA/Concern			I/Overlap		E/Concern
ImpRec	Change Task	PA/Issue Reports	P/Tokenize W/Complex	W. Vector	I/SS/Lucene, Search alg.	R/Score	Change Impacts (Unclear)

RSSE	**Category**	**Corpus**			**Recommendation Engine**		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
Suade	Change Task				I/W/DOI	R/Score	E/Components
Altair	API Use	SC/External Library	Heuristics	G/Augmented Access Graph	I/Overlap + Clustering	R/Score F/Threshold	C/API Methods +LL
MAPO	API Use	SC/Projects	Mining H/Clustering	Patterns	I/LS/Simple/ Levenshtein	R/Score	E/API Patterns
Strathcona	API Use	SC/Projects		S/BoW	I/LS/Simple	R/Score F/Threshold	LL/Code Snippets Graphical View Rationale
Heinemann and Hummel	API Use	SC/Projects	P/Tokenize	M/Binary	I/SS/Simple/ Hamming	R/Occurrences number	API Methods (Unclear presentation)
Selene	API Use	SC/Projects	P/Tokenize W/Complex	W. Vector	I/SS/Advanced	R/Score F/Threshold	LL/Code snippets
APISynth	API Use	SC/Projects	W/Simple Count	G/DAG	I/Search Algorithm	R/Path Length	LL/API Methods

RSSE	**Category**	**Corpus**			**Recommendation Engine**		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
Bavota et al.	Refactoring				NI/Threshold		Structural Properties
Thies and Roth	Refactoring				I/LS/Simple		Structural Properties
DCLFix	Refactoring				I/SS/Simple/ Jaccard		LL/Structural Properties
SemDiff	Refactoring	PA/Log	H/Clustering, Mining	Set	I/Weighting	R/Score F/Threshold	LL/Methods
Cordeiro et al.	Debugging	QA/Stack-Overflow	H/Mining	Code Snippets	I/SS + LS	R/Score F/Threshold	LL/Web Pages
SurfClipse	Debugging	Dynamic corpus/ search results			I/W/PageRank LS	F/Threshold R/Score	LL/Search Queries LL/Web Pages
Exception-Tracer	Debugging	SC/Projects QA/SO			I/Search algorithm		LL/Code snippets LL/Web Pages

RSSE	**Category**	**Corpus**			**Recommendation Engine**		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
De Souza et al.	Debugging	QA/Stack-Overflow	H/Mining, P/Tokenize, Stemming	Patterns	I/SS/Lucene	R/Score F/Threshold	LL/Documents (Q&A pairs)
Debug-Advisor	Debugging	PA/Log, Bug report	P/Tokenize P/Custom Feature W/TF-IDF Mining	S/BoW Relationship Graph	I/LS/Advanced I/W/PageRank	R/Score	LL/Methods, Bug Report
AutoFix	Debugging				NI/Binary tagging		LL/Code snippets
Sibyl	Debugging	PA/Bug Reports	H/Mining, Clustering, P/Tokenize, W/Complex	W. Vectors	I/SS/Advanced	R/Score F/Threshold	Drop-Down Menu/ Developers, Components
ScoreRec	Debugging				I/LS, Weighting	R/Score	LL/Methods + Rationale

RSSE	**Category**	**Corpus**			**Recommendation Engine**		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
Erfani et al.	Debugging	SC/Projects Unit Test Cases			NI/Threshold		Unit Test Cases (Unclear presentation)
Test Tenderer	Debugging	SC/Projects Unit Test Cases	P/Complex	Dependencies	I/Clustering	R/Custom Criteria	Test Cases (Unclear presentation)
Kpodjedo et al.	Debugging				I/Weighting		Graphical View
SPARS-J	Component	IH/ Browsing	Binary Tagging	Ratings	I/CF	R/Score F/Browsed Components	E/Software Components
Rascal	Component	SC/Projects	W/Simple Count	W. Matrix	I/CF	F/CBF R/Score	E/Software Components
Javawock	Component	SC/Projects	B. Tagging	B. Matrix	I/CF	R/Score	LL/Components
A-SCORE	Component	SC/Projects	P/Tokenization W/Simple Count, LSI	W. Matrix	I/SS	R/Score	E/Software Components

RSSE	**Category**	**Corpus**			**Recommendation Engine**		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
Code-Conjurer	Component	Dynamic corpus/ search results			NI/Simple Count	F/Threshold	E/Software Components and method signatures
Sando	Exploration	SC/Projects		Tree Matrix S/Term Pairs	I/LS/Simple		Drop-down Menu/ Search Queries
Refoqus	Exploration	Queries + Search Results	W/Complex LS, Clustering	Tree	I/SS/Lucene		Reformulation strategy (Unclear presentation)
Reverb	Exploration	IH/ Browsing			I/SS/Advanced	R/Score F/Threshold	LL/Web Pages
NavClus	Exploration	IH/ Browsing	H/Clustering	Patterns (clusters of sequences)	I/W/TF-IDF		Graphical View
Sora	Exploration				I/W/PageRank	F/Threshold	Files (Classes) (Unclear p.)

RSSE	**Category**	**Corpus**			**Recommendation Engine**		
		Raw Data	**Treatment** *H=Heuristic W=Weighting*	**Processed Data** *M=Matrix*	**Treatment** *LS=Lexical Similarity*	**Filtering/Ranking** *R=Ranking F=Filtering*	**Recommendations Nature** *LL=Links List*
McMillan et al.	Prototyping	PA/ Documents SC/Projects	H/Mining, Clustering	Matrix Graph	I/SS, CBF I/W/PageRank	F/Threshold	Check List/Features E/Packages
TagRec	Tagging				I/Weighting (complex)	F/Threshold	List of tags
Conscius	Experts	PA/Email, Javadoc, SC/Project	H/Mining	Dependencies	I/Weighting, SS/Advanced	R/Score	Experts (unclear presentation)

BIBLIOGRAPHY

Adomavicius, Gediminas and Alexander Tuzhilin. 2005. “Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions”. *IEEE Trans. Knowl. Data. En.* vol. 17, n° 6, (June 2005), p. 734-749.

Adomavicius, Gediminas and Alexander Tuzhilin. 2011. “Context-Aware Recommender Systems”. In *Recommender Systems Handbook,* Ricci, Francesco, Lior Rokach, Bracha Shapira and Paul B. Kantor (Eds.), p. 217-253. New York: Springer Science + Business Media.

Al-Kofahi, Jafar M., Ahmed Tamrawi, Tung Thanh Nguyen, Hoan Anh Nguyen and Tien N. Nguyen. 2010. “Fuzzy Set Approach for Automatic Tagging in Evolving Software”. In *Proceedings of the International Conference on Software Maintenance, ICSM,* (Timisoara, Romania, September 12-18, 2010), p. 1-10.

Amintabar, Vahid, Abbas Heydarnoori and Mohammad Ghafari. 2015. “ExceptionTracer: A Solution Recommender for Exceptions in an Integrated Development Environment”. In *Proceedings of the 23rd International Conference on Program Comprehension,* (Florence, Italy, May 18-19, 2015), p. 299-302.

Antunes, Bruno, Joel Cordeiro and Paulo Gomes. 2012. “SDiC: Context-based retrieval in Eclipse”. In *Proc. of the 34th ICSE (Zurich, Switzerland)*, p. 1467-1468.

Anvik, John and Gail C. Murphy. 2011. “Reducing the effort of bug report triage: Recommenders for development-oriented decisions”. *ACM Transactions on Software Engineering and Methodology,* vol. 20, n° 3, p 10 (35 pp.), Aug. 2011

Ashok, B., Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa and Vipindeep Vangala. 2009. “DebugAdvisor: A Recommender System for Debugging”. In *Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (Amsterdam, The Netherlands, August 24 - 28, 2009), ESEC-FSE '09, p. 373-382.

Avazpour, Iman, Teerat Pitakrat, Lars Grunske and John Grundy. 2014. “Dimensions and metrics for evaluating recommendation systems”. In *Recommendation Systems in Software Engineering,* Robillard, Martin P., Walid Maalej, Robert J. Walker and Thomas Zimmermann (Eds.), p. 245-273. Berlin: Springer-Verlag.

Bavota, Gabriele, Sebastiano Panichella, Nikolaos Tsantalis, Massimiliano Di Penta, Rocco Oliveto and Gerardo Canfora. 2014. “Recommending refactorings based on team Co-maintenance patterns”. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering,* (Vasteras, Sweden, September 15 - 19, 2014), ASE '14, p. 337-342.

Bazire, Mary and Patrick Brézillon. 2005. “Understanding context before using it”. In *Proceedings of 5th International Conference on Modeling and Using Context,* p. 29-40.

Borg, Markus. 2014. “Context-Based recommendation to support problem solving in software development”. In *Proceedings of 29th ACM/IEEE International Conference on Automated Software Engineering,* (Vasteras, Sweden, September 15-19, 2014), p. 891-894.

Cordeiro, Joel, Bruno Antunes, and Paulo Gomes. 2012. “Context-Based recommendation to support problem solving in software development”. In *Proceedings of Third International Workshop on Recommendation Systems for Software Engineering,* (Zurich, Switzerland), RSSE '12, p. 85-9.

Cubranic, Davor, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. 2005. “Hipikat: a project memory for software development”. *IEEE Trans. Software En,* vol. 31, n° 6, (June 2005), p. 446-65.

Czarnecki, Krzysztof, Chang Hwan Peter Kim and Karl Trygve Kalleberg. 2006. “Feature models are views on ontologies”. In *Proceedings of 10th International Software Product Line Conference,* p. 41-51.

Dagenais, Barthélémy and Martin P. Robillard. 2008. “Recommending adaptive changes for framework evolution”. In *Proceedings of the 30th International Conference on Software Engineering,* (Leipzig, Germany, May 10-18, 2008), p. 481-490.

Denninger Oliver. 2012. “Recommending relevant code artifacts for change requests using multiple predictors”. In *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering,* (Zurich, Switzerland, June 04, 2012), p. 78-79.

De Souza, Lucas B. L., Eduardo C. Campos and Marcelo de A. Maia. 2014. “Ranking crowd knowledge to assist software development”. In *Proceedings of the 22nd International Conference on Program Comprehension,* (Hyderabad, India, June 2-3, 2014), p. 72-82.

Dey, Anind K., Gregory D. Abowd and Daniel Salber. 2001. “A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications”. *Human-Computer Interaction,* vol. 16, (2001), p. 97–166.

Dourish, Paul. 2004. “What we talk about when we talk about context”. *Pers. Ubiquit. Comput.* vol. 8, n° 1 (2004), p. 19-30.

Erfani, Mostafa, Iman Keivanloo and Juergen Rilling. 2013. “Opportunities for Clone Detection in Test Case Recommendation”. In *Proceedings of 37th Annual Computer Software and Applications Conference Workshops,* p. 65-70.

Gall, Harald C., Beat Fluri and Martin Pinzger. 2009. “Change analysis with evolizer and changedistiller”. *IEEE Software,* vol. 26, n° 1 (January-February 2009), p. 26-33.

Ge, Xi, David Shepherdy, Kostadin Damevskiz and Emerson Murphy-Hill. 2014. “How the Sando Search Tool Recommends Queries”. In *Proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering,* (Antwerp, Belgium), CSMR-WCRE '14, p. 425-8.

Haiduc, Sonia, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia and Tim Menzies. 2013. “Automatic Query Reformulations for Text Retrieval in Software Engineering”. In *Proceedings of the 35th International Conference on Software Engineering,* (San Francisco, CA, United states, May 18-26, 2013), p. 842-851.

Happel, Hans-Jörg and Walid Maalej. 2008. “Potentials and challenges of recommendation systems for software development”. In *Proceedings of International Workshop on Recommendation Systems for Software Engineering, Co-located with the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering,* (November 10, 2008), RSSE '08, p. 11-15.

Heinemann, Lars and Benjamin Hummel. 2011. “Recommending API methods based on identifier contexts”. In *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation, Co-located with ICSE 2011,* (Waikiki, Honolulu, HI, USA, May 28, 2011). SUITE '11, p. 1-4.

Holmes, Reid, Robert J. Walker and Gail C. Murphy. 2006. “Approximate structural context matching: An approach to recommend relevant examples”. *IEEE Trans. Software Eng,* vol. 32, n° 12 (December 2006), p. 952-970.

Holmes, Reid, Tristan Ratchford, Martin P. Robillard and Robert J. Walker. 2009. “Automatically Recommending Triage Decisions for Pragmatic Reuse Tasks”. In *Proceedings of 24th IEEE/ACM International Conference on Automated Software Engineering,* (Auckland, New zealand, November 16-20, 2009), p. 397-408.

Hummel, Oliver, Werner Janjic and Colin Atkinson. 2010. “Proposing software design recommendations based on component interface intersecting”. In *Proceedings of 2nd International Workshop on Recommendation Systems for Software Engineering in Conjunction with the 32nd ACM/IEEE International Conference on Software Engineering,* (Cape Town, South Africa, May 04, 2010), RSSE '10, p. 64-68.

Ichii, Makoto, Yasuhiro Hayase, Reishi Yokomori, Tetsuo Yamamoto and Katsuro Inoue. 2009. “Software component recommendation using collaborative filtering”. In

Proceedings of ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, (Vancouver, Canada, May 16, 2009). SUITE '09, p. 17-20.

Janjic, Werner and Colin Atkinson. 2013. "Utilizing software reuse experience for automated test recommendation". In *Proceedings of the 8th International Workshop on Automation of Software Test,* (San Francisco, CA, United states, May 18-19, 2013), p. 100-106.

Kitchenham, Barbara, 2004. Procedures for Performing Systematic Reviews. Keele University Technical Report, Keele, UK.

Kpodjedo, Segla, Filippo Ricca, Philippe Galinier and Giuliano Antoniol. 2008. "Not all classes are created equal: Toward a recommendation system for focusing testing". In *Proceedings of International Workshop on Recommendation Systems for Software Engineering, Co-located with the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering,* (Atlanta, GA, United states, November 9, 2008), p. 6-10.

Lee, Seonah and Sunghun Kim. 2015. "The Impact of View Histories on Edit Recommendations". *IEEE Transactions on Software Engineering,* vol. 41, n° 3 (March 2015), p. 314-30.

Lee, Seonah, Sungwon Kang and Matt Staats. 2013. "NavClus: a graphical recommender for assisting code exploration". In *Proceedings of 35th International Conference on Software Engineering,* (San Francisco, CA, USA), ICSE '13, p. 1315-18.

Long, Fan, Xi Wang and Yang Cai. 2009. "API Hyperlinking via structural overlap". In *Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (Amsterdam, The Netherlands, August 24 - 28, 2009). ESEC-FSE '13, p. 203-212.

Lozano, Angela, Andy Kellens and Kim Mens. 2011. "Mendel: Source Code Recommendation based on a Genetic Metaphor". In *Proceedings of 26th IEEE/ACM Int. Conf. Autom.* (Lawrence, KS, USA). ASE '11, p. 384-7.

Lu, Jie, DianshuangWu, Mingsong Mao,Wei Wang and Guangquan Zhang. 2015. "Recommender System application developments: A survey". *Decision Support Systems*, vol. 74, p. 12-32.

Lv, Chen, Wei Jiang, Yue Liu, and Songlin Hu. 2014. "APISynth: A New Graph-Based API Recommender System". In *Proceedings of 36th International Conference on Software Engineering, ICSE Companion,* (Hyderabad, India, May 31 - June 07, 2014). ICSE Companion '14, p. 596-7.

Maki, Sana, Sègla Kpodjedo and Ghizlane El Boussaidi. 2015. “Context Extraction in Recommendation Systems in Software Engineering: A Preliminary Survey”. In *Proceedings of 25th Conference of the Center for Advanced Studies on Collaborative Research (CASCON),* (IBM Corp., Markham, Ontario, Canada, November 2 - 4, 2015), p. 151-160. Gould, Jordan, Marin Litoiu and Hanan Lutfiyya (Eds.).

Malheiros, Yuri, Alan Moraes, Cleyton Trindade and Silvio Meira. 2012. “A source code recommender system to support new comers”. In *Proceedings of the 36th Annual Computer Software and Applications Conference, (COMPSAC '12),* p. 19-24.

McCarey, Frank, Mel Ó Cinnéide and Nicholas Kushmerick. 2005. “Rascal: a recommender agent for agile reuse”. *Artificial Intelligence Review,* vol. 24, n° 3-4 (December 2005), p. 253-76.

McMillan, Collin, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang and Bamshad Mobasher. 2012. “Recommending Source Code for Use in Rapid Software Prototypes”. In *Proceedings of the 34th International Conference on Software Engineering,* (Zurich, Switzerland, June 2-9, 2012), p. 848-858.

Mens, Kim and Angela Lozano. 2014. “Source Code-Based Recommendation Systems”. In *Recommendation Systems in Software Engineering,* Robillard, Martin P., Walid Maalej, Robert J. Walker and Thomas Zimmermann (Eds.), p. 93-130. Berlin: Springer-Verlag.

Mkaouer, Wiem, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb and Mel Ó Cinnéide. 2014. “Recommendation System for Software Refactoring Using Innovization and Interactive Dynamic Optimization”. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering,* (September 15–19, 2014, Vasteras, Sweden.), p. 331-336.

Mohebzada, Jamshaid G., Guenther Ruhe and Armin Eberlein. 2012. “Systematic Mapping of Recommendation Systems for Requirements Engineering”. In *International Conference on Software and System Process, ICSSP 2012,* p. 200-209.

Moraes, Alan, Eduardo Silva, Cleyton da Trindade, Yuri Barbosa and Silvio Meira. 2010. “Recommending Experts Using Communication History”. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering,* (Cape Town, South africa, May 4, 2010) p. 41-45.

Murakami, Naoya, Hidehiko Masuhara and Tomoyuki Aotani. 2014. “Code Recommendation Based on a Degree-of-Interest Model”. In *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*, (Hyderabad, India, June 3, 2014), p. 28-29.

Niu, Nan, Fangbo Yang, Jing-Ru C. Chengy and Sandeep Reddivari. 2012. “A Cost-Benefit Approach to Recommending Conflict Resolution for Parallel Software Development”. In *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering*, (Zurich, Switzerland, June 4, 2012), p. 21-25.

Pakdeetrakulwong, Udsanee, Pornpit Wongthongtham and Waralak V. Siricharoen. 2014. “Recommendation systems for software engineering: A survey from software development life cycle phase perspective”. In *Proceedings of 9th International Conference for Internet Technology and Secured Transactions,* (Feb. 10, 2015). ICITST '14, p. 137-142.

Pei, Yu, Carlo A. Furia, Martin Nordio and Bertrand Meyer. 2015. “Automated Program Repair in an Integrated Development Environment”. In *Proceedings of 37th IEEE International Conference on Software Engineering*, (Florence, Italy, May 16-24, 2015), p. 681-684.

Proksch, Sebastian, Veronika Bauer and Gail C. Murphy. 2015. “How to Build a Recommendation System for Software Engineering”. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Software Engineering - International Summer Schools, LASER 2013–2014, Revised Tutorial Lectures,* vol. 8987, p. 1-42.

Rahman, Mohammad Masudur and Chanchal K. Roy. 2014. “SurfClipse: Context-Aware Meta-search in the IDE”. In *Proceedings of 30th International Conference on Software Maintenance and Evolution, ICSME '14,* p. 617-20.

Ricci, Francesco, Lior Rokach and Bracha Shapira. 2011. “Introduction to Recommender Systems Handbook”. In *Recommender Systems Handbook,* Ricci, Francesco, Lior Rokach, Bracha Shapira and Paul B. Kantor (Eds.), p. 1-35. New York: Springer Science + Business Media.

Robillard, Martin P. and Putra Manggala. 2008. “Reusing program investigation knowledge for code understanding”. In *Proceedings of 16th International Conference on Program Comprehension. ICPC '08,* p. 202-11.

Robillard, Martin P., Robert J. Walker and Thomas Zimmermann. 2010. “Recommendation Systems for Software Engineering”. *IEEE Software,* vol. 27, n° 4, (Juillet - Aout), p. 80-6.

Robillard, Martin P. and Robert J. Walker. 2014. “An Introduction to Recommendation Systems in Software Engineering”. In *Recommendation Systems in Software Engineering,* Robillard, Martin P., Walid Maalej, Robert J. Walker and Thomas Zimmermann (Eds.), p. 1-11. Berlin: Springer-Verlag.

Said, Alain, Domonkos Tikk and Paolo Cremonesi. 2014. “A Methodology for Ensuring the Relative Quality of Recommendation Systems in Software Engineering”. In *Recommendation Systems in Software Engineering,* Robillard, Martin P., Walid Maalej, Robert J. Walker and Thomas Zimmermann (Eds.), p. 275-300. Berlin: Springer-Verlag.

Sawadsky, Nicholas, Gail C. Murphy and Rahul Jiresal. 2013. “Reverb: Recommending Code-Related Web Pages”. *ICSE '13,* p. 812-21.

Shimada, Ryuji, Yasuhiro Hayase, Makoto Ichii, Makoto Matsushita and Katsuro Inoue. 2009. “A-SCORE: Automatic software component recommendation using coding context”. In *31st International Conference on Software Engineering - Companion Volume - ICSE-Companion*, (Vancouver, Canada, May 16-24, 2009), p. 439-40.

Sora Ioana. 2015. “A PageRank based recommender system for identifying key classes in software systems”. In *Proceedings of the 10th Jubilee IEEE International Symposium on Applied Computational Intelligence and Informatics*, (Timisoara, Romania, May 21-23, 2015), p. 495-500.

Terra, Ricardo, Marco Tulio Valente, Krzysztof Czarnecki and Roberto S. Bigonha. 2012. In *Proceedings of 16th European Conference on Software Maintenance and Reengineering,* (Szeged, Hungary, March 27-30, 2012), p. 335-340.

Thies, Andreas and Christian Roth. 2010. “Recommending rename refactorings”. In *Proceedings of 2nd International Workshop on Recommendation Systems for Software Engineering in Conjunction with the 32nd ACM/IEEE International Conference on Software Engineering,* (Cape Town, South Africa, May 04, 2010), RSSE '10, p. 1-5.

Thompson, C. Albert and Gail C. Murphy. 2014. “Recommending a starting point for a programming task: An initial investigation”. In *Proceedings of 4th International Workshop on Recommendation Systems for Software Engineering,* (Hyderabad, India, June 03, 2014), RSSE '14, p. 6-8.

Tsunoda, Masateru, Takeshi Kakimoto and Naoki Ohsugi. 2005. “Javawock: A java class recommender system based on collaborative filtering”. In *Proceedings of 17th International Conference on Software Engineering and Knowledge Engineering,* (Taipei, Taiwan, July 14-16, 2005), p. 491-497.

Yamada, Hiroaki and Atsuo Hazeyama. 2013. “A support system for helping to understand a project in software maintenance using the program package name”. In *Proceedings of 12th International Conference on Computer and Information Science,* (Niigata, Japan, June 16-20, 2013), p. 411-416.

Zagalsky, Alexey, Ohad Barzilay and Amiram Yehudai. 2012. “Example Overflow: Using Social Media for Code Recommendation”. In *Proceedings of Third International Workshop on Recommendation Systems for Software Engineering,* (Zurich, Switzerland), RSSE '12, p. 38-42.

Zhong, Hao, Tao Xie, Lu Zhang, Jian Pei and Hong Mei. 2009. “MAPO: Mining and Recommending API Usage Patterns”. *ECOOP '09. ACM, New York, NY,* p. 318-343.