What Recommendation Systems for Software Engineering Recommend: A Systematic Literature Review

Article *in* Journal of Systems and Software · November 2015

DOI: 10.1016/j.jss.2015.11.036

2 authors:

Marko Gašparič
Free University of Bozen-Bolzano
14 PUBLICATIONS **132** CITATIONS
SEE PROFILE

Andrea Janes
Free University of Bozen-Bolzano
80 PUBLICATIONS **1,075** CITATIONS
SEE PROFILE

What Recommendation Systems for Software Engineering Recommend: A Systematic Literature Review

Marko Gasparic and Andrea Janes
Free University of Bolzano
Emails: marko.gasparic@stud-inf.unibz.it, ajanes@unibz.it

Abstract

Background. A recommendation system for software engineering (RSSE) is a software application that provides information items estimated to be valuable for a software engineering task in a given context.
Objective. Present the results of a systematic literature review to reveal the typical functionality offered by existing RSSEs, research gaps, and possible research directions.
Method. We evaluated 46 papers studying the benefits, the data requirements, the information and recommendation types, and the effort requirements of RSSE systems. We include papers describing tools that support source code related development published between 2003 and 2013.
Results. The results show that RSSEs typically visualize source code artifacts. They aim to improve system quality, make the development process more efficient and less expensive, lower developer's cognitive load, and help developers to make better decisions. They mainly support reuse actions and debugging, implementation, and maintenance phases. The majority of the systems are reactive.
Conclusions. Unexploited opportunities lie in the development of recommender systems outside the source code domain. Furthermore, current RSSE systems use very limited context information and rely on simple models. Context-adapted and proactive behaviour could improve the acceptance of RSSE systems in practice.

Keywords: recommendation system for software engineering, systematic literature review

1. Introduction

Many software engineering techniques support the development of high-quality software, but the effort they require and the costs of learning them and applying them productively can be high [22].

To overcome such difficulties, the software engineering community develops tools that support the software engineer in her task. One type of such tools aims to suggest the "best" items according to the user's needs. The approach is similar to the one used in search engines and e-commerce recommendation systems[1], however, recommendation systems are being used for software engineering purposes only since recently [59].

1.1. Goal of this work

In this paper, we present the results of a systematic literature review we performed to explore an emerging research field of software engineering: recommendation systems for software engineering (RSSE). The goal of this work is to analyze which systems have been developed in the scientific community and how they can be used. Based on the results, we identified research gaps and possible directions for the future research.

Every (useful) information system processes some form of input and generates an output. The type of required input and the type of produced output have an influence on the acceptance of a given tool or method. Venkatesh et al. identify four aspects that "play a significant role as direct determinants of user acceptance and usage behavior" [66]:

1. Performance expectancy: the degree to which an individual believes that using the system will help him or her to attain gains in job performance.
2. Effort expectancy: the degree of ease associated with the use of the system.
3. Social influence: the degree to which an individual perceives that important others believe he or she should use the new system.
4. Facilitating conditions: the degree to which an individual believes that an organizational and technical infrastructure exists to support use of the system.

Based on these aspects, we were interested to study the literature to extract data about the first two determinants of acceptance and use: performance expectancy, i.e., the produced output and the therefore expected benefit, as well as effort expectancy, i.e., the expected effort to provide the required input. The first two aspects (performance and effort expectancy) are highly influenced by the way how the recommendation tool is conceived, while the last two aspects (social influence and facilitating conditions) are less impacted by the technical decisions behind the tool but by the context in which it is used: the technical infrastructure, social, and organizational aspects.

Furthermore, we were interested in how such tools are used during software engineering, more precisely, during the creation or modification of source code, which represents the core activity of software engineers to satisfy customer requirements [13]. Other software engineering activities, e.g., user requirements

[1]One of the most known examples of software that suggests to users what to do is the recommendation feature of Amazon.com, which recommends users what to buy, based on their past acquisitions.

management, can be also supported by recommendation systems, but we excluded such systems from our considerations since we wanted to focus on the core activity of handling source code.

Using the Goal-Question-Metrics template proposed by [11, 12], we formulate the research goal as follows:

Object of study:	Study recommendation systems for software engineering
Purpose:	to characterize them
Focus:	with respect to the produced recommendation, the benefits of this recommendation for the user, the required input, and the required effort to provide the input by the user
Viewpoint:	from the view point of the software engineer
Context:	in the context of tasks that involve the use of source code

In this study we do not distinguish between different software engineering roles (e.g., software design, coding, testing) as we are interested to obtain an overview of what recommendation systems can achieve and what kind of effort they expect in the context of the creation and maintenance of source code.

We formulated the following research questions "characterizing relevant attributes of the object of study with respect to the focus [11]":

1. What output do RSSE systems produce?
2. Which benefits does a software engineer get from using RSSE systems?
3. What input data do RSSE systems require?
4. Which effort does a RSSE system require by the software engineer?

The outcome of this systematic literature review can help practitioners and researchers to obtain an overview of how RSSE systems[2] are used and which research challenges still exist.

1.2. Definition of the term "recommendation system for software engineering"

According to [29], RSSEs provide development information (source code, artifacts, quality measures, and tools) and collaboration information (people, awareness, and status and priorities). After obtaining and evaluating relevant decisions they help software engineers to find relevant information [57]. We are using the following definition of a recommendation system for software engineering (RSSE):

[2]The term "RSSE system" is a tautology since the word "system" is already contained in the abbreviation "RSSE". Nevertheless, we use the term since it was introduced by the founders of this line of research in [57] and is used as well by its research community.

"A recommendation system for software engineering is a software application that provides information items estimated to be valuable for a software engineering task in a given context [57]."

2. Related Work

We are aware of two papers ([29] and [57]) and a book ([58]) in which the field is described and state-of-the-art tools are presented. Happel and Maalej [29] made a short survey of recommendation systems in software development, which includes six tools. They found that existing tools are focusing on "you might like what similar developers like" scenarios, and they encouraged researchers to focus on two aspects: what to recommend and when to recommend.

In 2010, Robillard et al. [57] published a short overview of the RSSE field, where they also presented the definition which emerged as a standard definition of RSSE systems. They focused on discussing what RSSEs do for developers, what their design dimensions are, and what the limitations and potentials of RSSE systems are. In the examples, Robillard et al. mentioned eight tools. They emphasized the "cold-start problem"[3], the need for recommendations explanations, and the lack of variance in the output, since RSSEs mostly recommend source code. They recognized proactive tools and tools that can adapt to user's feedback as promising directions for RSSE development.

The book "Recommendation Systems in Software Engineering" [58] presents the current research on RSSE systems, focusing on system design, implementation, and evaluation aspects of RSSEs. It is a comprehensive collection of techniques and approaches used in the field, without a particular focus on existing tools.

We are not aware of any comprehensive and systematic review of RSSE tools. With this SLR we fill the gap and summarize implemented RSSE systems that were presented in scientific journals and conferences.

3. Methodology

This section summarizes the methodological approach we used to identify the first set of papers and to filter out the relevant papers for our research questions.

We adopted the approach proposed by Kitchenham [37] for conducting software engineering SLRs; it is used to identify, evaluate and interpret all available research relevant to a particular research question, or to a topic area, or to a phenomenon of interest. It is recognized and commonly used by the software engineering community ([2, 3, 20, 28, 50, 60, 63]).

Following Kitchenham's guidelines, we defined a protocol before the execution of the SLR and followed it during the execution. The protocol contains

[3] "Cold-start problem" denotes a situation in which a recommender system does not have enough data (yet) to make quality recommendations.

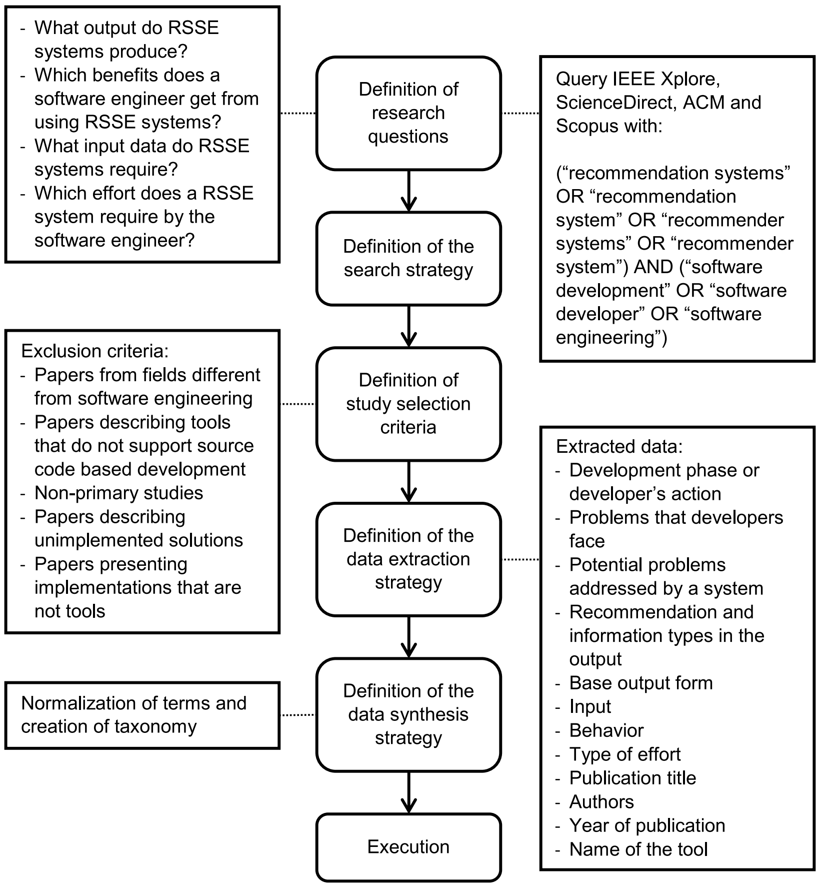
- What output do RSSE systems produce?
- Which benefits does a software engineer get from using RSSE systems?
- What input data do RSSE systems require?
- Which effort does a RSSE system require by the software engineer?
Definition of research questions
Query IEEE Xplore, ScienceDirect, ACM and Scopus with:
("recommendation systems" OR "recommendation system" OR "recommender systems" OR "recommender system") AND ("software development" OR "software developer" OR "software engineering")
Definition of the search strategy
Exclusion criteria:
- Papers from fields different from software engineering
- Papers describing tools that do not support source code based development
- Non-primary studies
- Papers describing unimplemented solutions
- Papers presenting implementations that are not tools
Definition of study selection criteria
Extracted data:
- Development phase or developer's action
- Problems that developers face
- Potential problems addressed by a system
- Recommendation and information types in the output
- Base output form
- Input
- Behavior
- Type of effort
- Publication title
- Authors
- Year of publication
- Name of the tool
Definition of the data extraction strategy
Normalization of terms and creation of taxonomy
Definition of the data synthesis strategy
Execution

Figure 1: Research process.

research questions, the search strategy, the inclusion and exclusion criteria, the data extraction strategy, the synthesis strategy, the study quality assessment, and the project timetable. Figure 1 depicts the process, which we describe below.

On 12th April 2013 we queried four digital libraries: IEEE Xplore[4], ScienceDirect[5], the ACM Digital Library[6], and Scopus[7]. We selected these libraries, because they include all high quality software engineering journals and proceedings of conferences.

To formulate the actual queries to extract the data from the selected digi-

[4]IEEE Xplore, `http://ieeexplore.ieee.org`
[5]ScienceDirect, `http://www.sciencedirect.com`
[6]ACM Digital Library, `http://dl.acm.org`
[7]Scopus, `http://www.scopus.com`

tal libraries, we framed the research questions using the PICOC (Population, Intervention, Comparison, Outcome, Context) criteria as suggested by [55] and [39]:

- Population: software development activities.
- Intervention: the use of recommendation systems to support the creation or modification of source code.
- Comparison: the situation in which the software engineer creates or modifies source code without the support of recommendation systems.
- Outcomes: characterization of the produced output, the therefore expected benefit, and the expected effort to provide the required input.
- Context: any context in which software engineers create or modify source code.

We used the following query to obtain all results for the population and the intervention:

> (”recommendation systems” OR ”recommendation system” OR ”recommender systems” OR ”recommender system”) AND (”software development” OR ”software developer” OR ”software engineering”)

To maximize recall, we did not include the comparison, outcomes, and context criteria in the search term. We consider the context criterion in exclusion criteria 1 and 2, while the outcomes and the comparison criteria defined our data extraction strategy (the comparison criterion defines what we consider the benefit of using a recommender system in software engineering).

IEEE Xplore returned 139, ScienceDirect 2, Scopus 189, and ACM 23 search results. We were interested only in the publications describing recommendation systems related to software development, particularly the ones that support source code development phases. We defined the following exclusion criteria:

1. Papers from fields different from software engineering, e.g., e-commerce or health-care recommendation systems.
2. Papers describing tools that do not support source code based development, e.g., tools supporting a requirements elicitation or a division of work.
3. Non-primary studies.
4. Papers describing unimplemented solutions, e.g., a new approach or a new algorithm.
5. Papers presenting implementations that are not tools, e.g., reusable source code libraries.

The exclusion criterion 3 is a typical feature of systematic literature reviews, i.e., systematic literature reviews assess primary studies [38]. The exclusion criteria 4 and 5 are based on the quality consideration that we wanted to consider only papers where the authors present already implemented recommendation systems and not hypothetical ones or only support tools for recommendation systems.

Out of 353 search results, 260 were unique. We excluded 32 results that were not papers published in journals or conferences (labeled "not studies" in Figure 2). Afterwards, based on reading the title and the abstract of the papers, we excluded 86, based on the defined exclusion criteria (they all matched the first exclusion criterion).

Out of 142 publications that were left for the full-text reading, we could not access 9. According to [37], in a systematic literature review it is necessary to avoid language bias, but we could not satisfy this condition, since we were only able to analyze papers in English. This resulted in exclusion of 2 other publications.

After reading and analyzing 131 papers, we obtained 46 relevant papers, that contained descriptions of 43 unique tools. The duplicates were [5, 6, 51, 52, 64, 65]. Figure 2 shows the filtering approach we used.

We prepared digital forms to accurately record any information needed to answer the research questions. We extracted these data categories:

1. Alternatively, the development phase or developer's action:
   - development phase: the main development phase that the tool supports, e.g., the implementation phase;
   - developer's action: a specific developer's action that is used independently from the phase or as a part of the phase, e.g., the search for a reusable component;
2. Problems that developers are facing: issues that developers want to address;
3. Potential problem addressed by the system: the authors' claim of what will happen if the suggested tool is not used. This aspect is related to the previous aspect 2;
4. Recommendation and information types in output: the items presented to the developer;
5. Base output form: how the recommendations are visualized. They can be visualized in different ways and they can be interactive. We were interested only in the initial output;
6. Input: all artifacts needed to produce a recommendation;
7. Behavior: the behavior of the system is either reactive or proactive (see section 4.4);
8. Type of effort: prerequisites that a developer has to explicitly satisfy, to make the system work or to obtain the recommendations[8];
9. Publication title;
10. Publication authors;

[8]We are only interested in actions that are performed to use an RSSE. Actions that would be performed regardless of a usage of a RSSE, e.g., programming, are not taken into account.

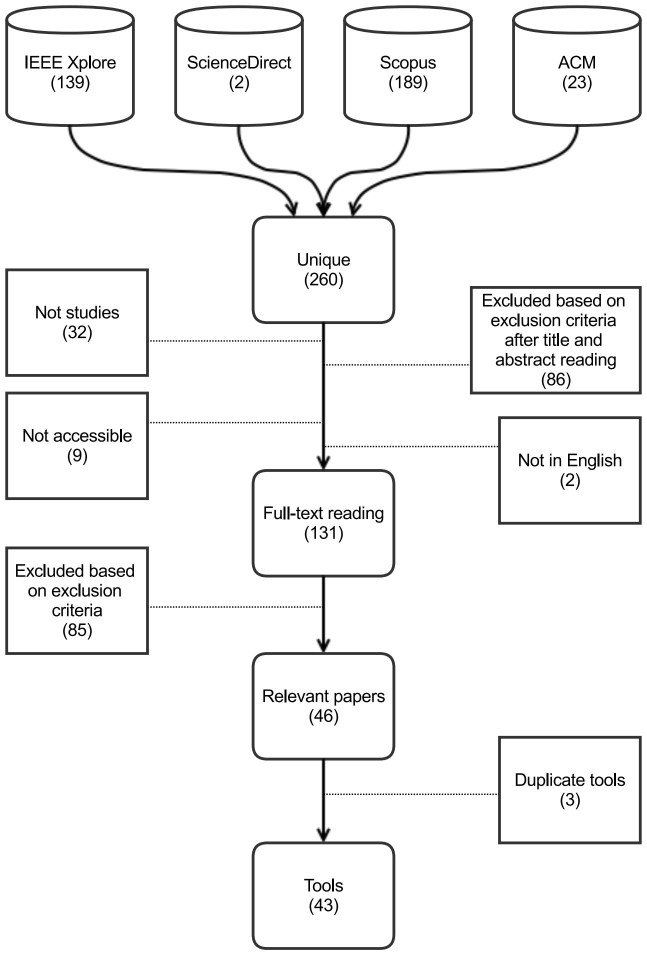

Figure 2: Filtering approach to filter 43 tools out of 260 papers.

11. Publication year;
12. Name of the tool.

During the synthesis phase, we normalized the terms describing the same phenomenon and we continued to use the most common term. We built a taxonomy using these terms. Since, due to time constraints, each paper was read only by one author and, consequentially, we could not evaluate extraction accuracy, we sent emails to all authors of the papers, asking if they are willing to evaluate the accuracy of the (normalized) extracted data related to their paper. 10 authors of the papers were willing to collaborate, and, as discussed in section 5, the accuracy was high. The subsequent merge of basic actions and phases, input and output types, and required type of effort into bigger groups was pair-reviewed by the two authors of this paper.

4. Results and Analysis

This section describes the results of our study; we discuss separately for each research question the answers we found in the identified literature[9].

4.1. Research question 1: What output do RSSE systems present to the user and how?

We identified different types of information and recommendations that RSSE present to their users and grouped them into 20 categories as in Table 1.

We further classified the found RSSE systems following the classification of [23] into product, process, or resource RSSEs:

- If the different papers describe an RSSE that describes, criticizes, or recommends source code or any other type of artifact that is handled during the software development process, we classified the RSSE as a "Product RSSE". Product RSSEs generate recommendations with the intention to use the recommendation during the creation of source code.
- If the RSSE describes, criticizes, or recommends a process to edit artifacts, we classified it as "Process RSSE".
- Finally, if the described RSSE describes, criticizes, or recommends a human or technical resource that might be useful, we classified the paper as "Resource RSSE".

Table 1 reports the output generated by the analyzed recommendation systems. According to our classification, the majority of RSSEs we found recommend source code, e.g., source code that needs to be changed or source code that might be interesting to reuse. Not only source code is recommended, also other files, either digital documents that might be interesting or binary files that represent deployed source code.

Table 1: Output generated by the analyzed recommendation systems.

Output	**Publications**	**Product**	**Process**	**Resource**
Binary source code files	[8]	×		
Changes in the deployment environment	[32]	×		
Design patterns	[53]	×		
Digital documents that might be interesting for the software engineer	[4]	×		

(Continues on next page)

[9]The used extracted raw data is publicly available at: `http://sites.google.com/site/mgasparic/slr`

Table 1, continued from the previous page

Output	Publications	Product	Process	Resource
Estimated artifact costs	[31]		×	×
Estimated defect density	[42]	×		
Help to perform changes	[15, 40, 64, 65, 51, 52]		×	
Messages posted on software engineering forums	[9, 16, 64, 65]	×		
Method parameters present in the API	[72]	×		
Necessary source code changes due to a previous modification	[24, 43]	×	×	
Possible experts, mentors, and collaborators	[8, 14, 15, 42, 47, 61]			×
Problem reports	[27, 64, 65]	×	×	
Reusable test steps	[41]	×	×	
Source code churn	[42]	×	×	
Source code that might be interesting for the software engineer (for different development environments, for different development phases, for different types of source code, for different types of requirements, or for different types of users)	[5, 6, 8, 10, 18, 19, 30, 31, 33, 36, 44, 45, 46, 48, 54, 56, 62, 64, 65, 68, 71]	×		
Source code that needs to be changed	[7, 17, 35, 34]	×		
Source code transformations	[21]	×		
Suggested improvements to the source code	[49]	×		
Suggested tools	[67]	×	×	×
Visited methods	[56]		×	
Words extracted from documentation and source code	[56, 68]	×		

Figure 3 summarizes the distribution of papers according to their classification into “product”, “process”, or “resource” (the classes are overlapping, i.e., articles can belong to more than one class; we count papers only once per class).

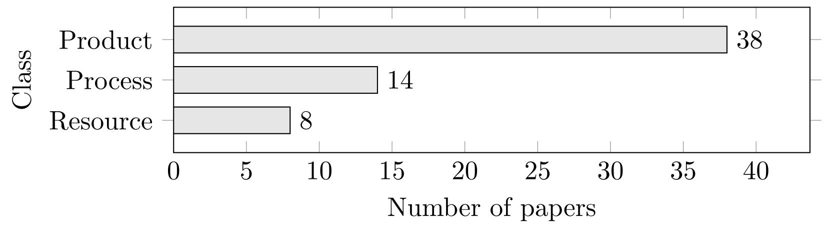

Figure 3: Distribution of obtained papers according to their classification into “product”, “process”, or “resource”.

Fourteen papers describe recommendations about the process, e.g., lists that help to perform complex changes, reusable testing steps, or the visited methods.

Finally, eight papers recommend human or technical resources that might be useful to the software engineer. This includes experts, mentors, or tools that can be consulted during development.

If we look at *how* recommendation systems present their results, thirty tools present recommendations in a list, [9] and [32] present them in a table, and [53] shows only one result. Five tools output documents; [36] uses documentation documents and adds source code examples to them, [18] creates a plan, [27] and [34] create a report, and [54] generates a source code. [34] outputs also dashboards. [40] and [42] output charts; [40] visualizes a change guide graph, while [42] visualizes a sliced circular view. [47] does not produce a direct output. We could not determine an output form of [15] and [61].

Table 2 shows the different types of outputs of the identified systems.

Table 2: Types of output produced by the identified systems.

Type of output	**Publications**
Not explained in the paper	[15, 61]
Change guide graph	[40]
Dashboard	[34]
Document	[18, 27, 34]
Documents with source code examples	[36]
List	[4, 5, 6, 7, 8, 10, 14, 16, 17, 19, 21, 24, 30, 31, 33, 35, 41, 43, 44, 45, 46, 48, 49, 51, 52, 56, 62, 64, 65, 67, 68, 71, 72]
No direct output	[47]
One result	[53]
Sliced circular view	[42]
Source code files	[54]
Table	[9, 32]

4.1.1. Discussion

From Table 2 we can see that the five-year-old statement of Robillard et al. [57] still holds: the predominant output mode of RSSEs is a simple recommendation list containing source code artifacts. However, we detected also some initiative in the community to develop RSSE tools that use different output forms, as well as innovative representations.

4.2. Research question 2: Which benefits does a software engineer get from using RSSE systems?

The authors of the systems claim that their systems can provide different benefits to software engineers. Since the systems are targeting highly diverse problems and most of them are limited to the specific kinds of actions or development phases, we analyzed the differences from three different aspects: which phase or action type is supported by the system, which actual problem the software engineer is facing, and which potential problem is prevented by the system.

As can be seen in Figure 4, the tools we identified mainly support for the implementation and maintenance phase (the classes are overlapping, i.e., articles can belong to more than one class; we count papers only once per class). As "implementation" we consider a development phase in which mainly new functionality is created, as "maintenance" a phase in which mainly existing code is modified, and as "information gathering" we classified papers that support learning activities, help a newcomer joining the team, help to find experts, code examples, or to navigate through various pieces of information in the IDE. We classified those papers that describe their tools as independent from a specific development phase as "phase independent".

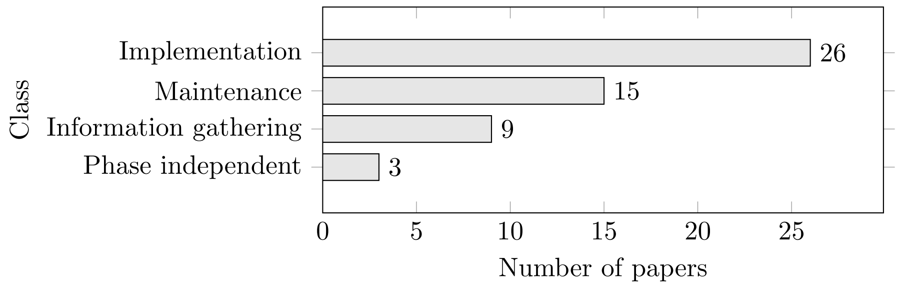

Figure 4: Distribution of obtained papers according to the development phase they support.

Twelve tools are specifically targeting reuse actions, four are targeting debugging actions, and four are helping newcomers that are joining the team.

According to the tools we identified, RSSEs mainly support reuse, debugging, implementation, and maintenance phases/activities. The identified RSSEs attempt to support the improvement of system quality, e.g., bug prevention and performance improvement, make development more efficient or less expensive, lower the cognitive load, and help the software engineer to take better decisions.

In Table 3 we report the problems that RSSEs address or which trigger the activation of an RSSE. Looking at this table, we see that the most common reason for a usage of a RSSE is a lack of knowledge by the software engineer. Moreover, many tools were developed because the existing tools lack a required functionality or do not perform the given functionality with the necessary quality.

Table 3: Systems addressing a certain problem that the software engineer is facing.

Problem that the software engineer is facing	**Publications**
Ad-hoc approach	[51, 52]
All dependencies have to be tracked	[40]
All requirements have to be met	[18]
Architecturally-relevant code anomalies are not detected soon enough	[7]
Changes have to be introduced	[24, 27]
Changes have to be tracked	[17, 32]
Clone removal is needed	[34]
Software engineer does not have enough knowledge	[8, 9, 16, 33, 36, 41, 42, 45, 47, 48, 53, 54, 67, 68, 72]
Difficult to search due to big data sets	[4, 5, 6, 45, 56, 64, 65]
Function was not designed to be reused as needed	[31]
Insufficient documentation	[4, 36, 43]
Mentoring is needed	[14, 44, 64, 65]
No focus on exception handling	[10]
Possible solutions have to be compared	[71]
Reoccurring similar actions	[21]
Reusable components missed	[35, 46, 62]
Insufficient help from existing tools	[15, 19, 27, 33, 40, 49, 54, 61, 64, 65, 71, 72]
Using multiple facts is more efficient	[30]

Table 4 shows which problems RSSEs aim to prevent. We can see that the most of the tools aim at improving the efficiency of the software engineers.

Table 4: Systems preventing a certain problem.

Potential problem addressed by a system	**Publications**
Big cognitive load for software engineer	[15, 18, 31, 56]
Bugs are introduced in the system	[10, 27, 40]
Complete redesign of the system is needed	[7]
Software engineer leaves work environment	[9, 16]
Current practice is expensive	[44, 51, 52, 72]
Current practice is inefficient	[4, 5, 6, 8, 17, 19, 21, 27, 30, 32, 33, 34, 36, 41, 43, 45, 46, 47, 48, 49, 51, 52, 56, 62, 67, 68, 72]
Lower quality of the system	[35, 45, 62]
Project delay or failure	[14]
Wrong decision taken	[42, 53, 54, 61]

(Continues on next page)

Table 4, continued from the previous page

Potential problem addressed by a system	Publications
System performance issues	[10]
Task is unfeasible	[64, 65, 71]
Usefulness of the system decreases	[24]

4.2.1. Discussion

Most of the systems are specialized to support debugging, maintenance, and implementation phases; reuse is well supported by existing tools as well. A comprehensive support for the testing phase is not present; we find it surprising, since it is one of the core phases in traditional as well as in newer software development approaches such as Lean or Agile. Otherwise, the existing RSSEs address relatively diverse range of kinds of potential development problems, from the process and product perspective.

4.3. Research question 3: What input data do RSSE systems use?

To answer research question 3, we analyzed the identified papers to understand how the output was generated, i.e., which data is used by the different approaches.

The results are shown in Table 5. As for the outputs, we followed again the classification of [23] and classified the inputs as “Product inputs” if the recommendation system uses source code or any other type of artifact that is handled during the software development process, as “Process inputs” if the recommendation system uses data about the software development process, and as “Resource inputs” if the recommendation system uses data about the employed human or technical resources.

Table 5: Inputs used by the analyzed recommendation systems.

Input	Publications	Product	Process	Resource
A list of known design patterns or components	[33, 35, 53, 62]	×		
A list of known tools	[67]			×
Communication between software engineers	[14, 15, 47, 64, 65]	×	×	×
Issue tracking system data	[8, 15, 19, 24, 42]	×		
Log files	[8, 16]	×		
Meta-data (e.g., annotations, meta models, documentation)	[4, 15, 21, 34, 36, 46, 47, 54, 56]	×		
Output generated by external systems	[36, 49]	×		

(Continues on next page)

Table 5, continued from the previous page

Input	Publications	Product	Process	Resource
Publicly accessible source code examples (e.g, in blogs, forums, repositories)	[9, 16, 45, 46, 48, 68, 71]	×		
Software development process (actions, their occurrence, sequence, and duration)	[5, 6, 21, 30, 33, 40, 56, 64, 65, 67]		×	
Source code (in the development environment or present in software revision control systems, e.g., CVS[10] or SVN[11])	[5, 6, 8, 10, 14, 15, 17, 19, 21, 24, 27, 30, 32, 34, 36, 41, 42, 43, 44, 45, 47, 48, 51, 52, 54, 56, 61, 62, 64, 65, 67, 72]	×		
Tasks (e.g., user stories, bug reports)	[41, 44, 64, 65, 67]	×	×	×
Test artifacts	[27, 34, 41]	×		
The deployed system	[35]	×		
User input (e.g., search terms, a query, request for help, settings, preferences)	[4, 5, 6, 7, 8, 9, 14, 17, 18, 27, 30, 33, 36, 46, 51, 52, 53, 54, 64, 65, 68, 71]	×	×	×

Figure 5 summarizes the distribution of papers according to their classification into "product", "process", or "resource" (the classes are overlapping, i.e., articles can belong to more than one class; we count papers only once per class).

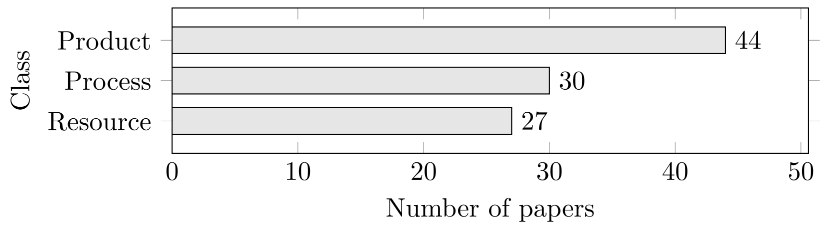

Figure 5: Distribution of obtained papers according to the input type they require.

We identified fourteen different input types; the most common are source code and manual user input. Ten papers monitor the software development process, i.e., actions that the software engineers perform, when these actions occur, the sequence of these actions, and how long they take.

[10]http://www.nongnu.org/cvs/
[11]http://subversion.apache.org

4.3.1. Discussion

The type of input and the frequency with which it is read have an impact on the analysis of the collected data as well as on the development effort of a particular RSSE. For instance, source code is easier to parse and analyze than natural language. Almost all existing RSSEs take project artifacts as an input data. Because RSSEs cannot crawl into developer's head to obtain the exact context [57], they try to use the artifacts that are generated during development. Few systems from our review also observe developer's actions, e.g., navigation patterns or the time and duration of certain actions, or reactions, e.g., user's feedback. In this way, RSSEs can only create general estimations of context, but other approaches would likely require additional developer's effort.

4.4. Research question 4: Which effort does a RSSE system require by the software engineer?

We identified the behaviour of the identified systems and we assessed the level of manual effort that the systems require to function as expected.

A RSSE is reactive if the user has to explicitly make a call to obtain the recommendations. A proactive system — the opposite of a reactive — presents recommendations automatically; in this way software engineers do not have to realize themselves that they need recommendations [58]. Table 6 reports the classification of all papers into these two groups.

Table 6: Systems using a certain behavior.

Behavior	**Publications**
Proactive	[5, 6, 16, 24, 27, 32, 34, 40, 42, 43, 45, 48, 56, 62, 67]
Reactive	[4, 7, 8, 9, 10, 14, 15, 17, 18, 21, 30, 31, 33, 34, 35, 36, 41, 44, 46, 47, 49, 51, 52, 53, 54, 64, 65, 68, 71, 72]
Not explained in the paper	[19, 61]

Figure 6 summarizes the distribution of papers of Table 6 (the classes are overlapping, i.e., articles can belong to more than one class; we count papers only once per class).

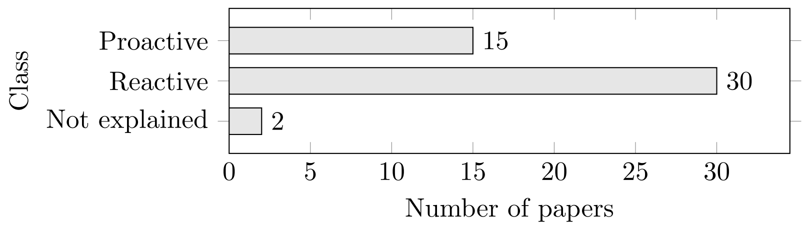

Figure 6: Distribution of obtained papers according to their behaviour classification.

We could not identify the classification of [10, 19, 61], but the author of [10] informed us that it is a reactive tool. In summary, we identified twenty-seven reactive tools, thirteen proactive tools, and one tool that can be used in both ways.

Conscius [47] is the only tool that does not present any recommendation directly. When a software engineer submits a message that has to be addressed by an expert, the tool automatically forwards it to a recommended expert. Since software engineers are aware of effects, we can treat their action as an explicit launch of a recommendation process, which makes the system reactive.

In Figure 7 we visualize what kind of effort is required by the software engineer; "extensive effort" would, in our opinion, require a change in development process, and "low effort" requires submitting an explicit request or filling out a short form, e.g., query submission.

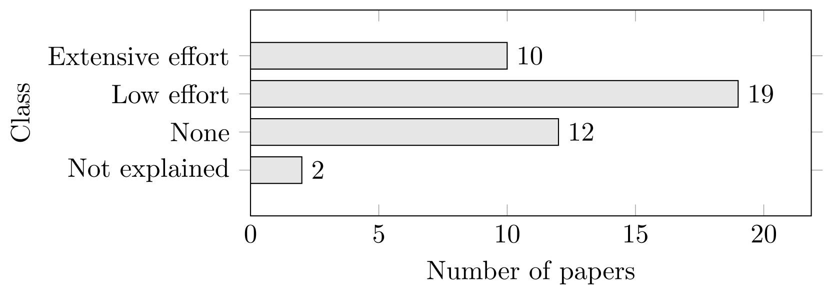

Figure 7: Number of systems requiring a certain level of manual effort to function.

We could not identify the required effort of seven systems, but we concluded that five of them require at least running a program, which we consider equivalent to submitting a request. Another five tools require to submit a request, and one requires either to submit a request or to formulate a query. Seven tools require only to submit a query, and one tool requires to submit a message, which is delivered to the recommended recipient. Twelve tools, which are all proactive, do not require any effort. All other tools require a non-negligible effort, compared to a query submission.

A prerequisite of the tool described in [18] is the definition of formal contexts. [54] and [21] require manually written annotations in source code; [54] also demands the generation of a predefined report that translates each automatically generated diagram into a PROLOG predicate. A merging scope has to be defined when scoreRec [51, 52] is used. In SCOOP [7] the user has to set thresholds and detection strategies that are used by the tool. Conscius [47] expects the software engineer to submit a message. In DebugAdvisor [8] an expert has to write a feature parser and submit a query to generate recommendations. The tool described in [46] reacts when a product description is inserted. In DPR [53] the user has to give answers to questions formulated by the tool. [15] expects issue insertion or search execution.

A RSSE can be proactive even if it requires manual effort to function as expected. However, it is uncommon: out of fourteen proactive tools, we identified two such systems. In ASPEN [27], a user has to define a model, which is later used together with a source code and test artifacts during a recommendation process. Nevertheless, it is the only standalone proactive system we identified: it mainly runs in a background, and because it generates recommendations without an explicit request (after it is started), we classified it as proactive. In Spyglass [67] a user has to select an icon to visualize recommendations. This is another exceptional case, since the submission of a request is a representative aspect of reactive tools. However, Spyglass generates recommendations automatically and with a change of an icon only informs the user that recommendations exist.

4.4.1. Discussion

Less than a third of the tools demand non-negligible effort by the software engineer, while more than a quarter do not demand any manual effort; others expect that the user issues a request (we call a “request” the click on a button or the execution of a menue command) or formulates a query (we call a “query” a request for information which contains hints on how to retrieve the information) to the RSSE.

It is not surprising that most of the systems are reactive, since the design and the implementation are generally easier; it is difficult not to cross the line between providing helpful recommendations and spamming. Additionally, standalone tools are reactive almost by the definition; one exception is described in the previous section. What we find surprising is that reactive systems are increasingly popular, this can be seen also in Figure 8. We expected that an advancement in the research will encourage the development of proactive systems, but our expectations were wrong.

5. Validity Threats

We used the list of threats defined by Wohlin et al. [70], which includes four types of validity threats, namely, conclusion, internal, construct, and external validity threats.

5.1. Conclusion validity

Conclusion validity in qualitative studies represents reliability of conclusions drawn from the collected data [70]. In our case, threats are related to potentially too small samples and violations of assumptions.

The used methodology [37] already assumes that not all relevant primary studies that exist can be identified. As tactics, we lowered the effect of this validity threat with the selection of a query with which we achieved a high recall, at the cost of a low precision. Nevertheless, due to publication bias, positive results are more likely to be published than negative, which may cause that certain relevant primary studies were not published. We were not able to address this threat.

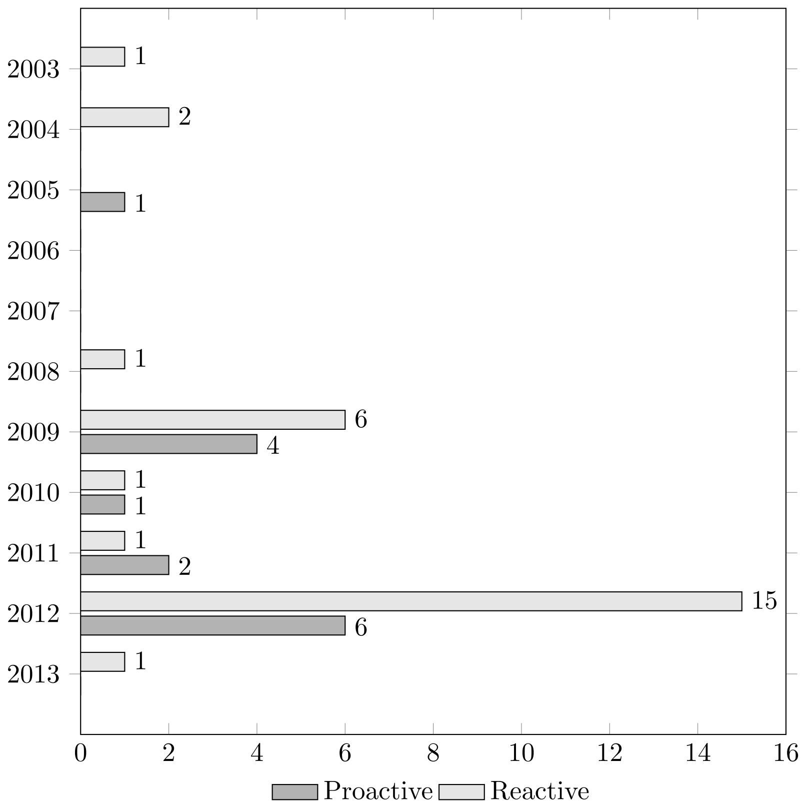

Figure 8: Distribution of the developed RSSEs over the years.

According to [37], duplicated reports have to be avoided since they could bias the results; the most recent report has to be taken into account. In one case, we obtained three pairs of publications describing the same tool. All of them have the same authors and the extracted data coincided. The author of one pair of papers approved that they are describing the same tool, but the others did not reply to the survey. We did not exclude the older publication, but we merged the results and treated them as one publication.

5.2. Internal validity

Internal validity threats are related to possible wrong conclusion about causal relationships between treatment and outcome [70]. In the case of a literature review, internal validity describes how well the findings represent the true opinion expressed in the literature. The primary objective of a *systematic* literature review, such as the present one, is to minimize internal validity threats. To address validity threats, researchers use "tactics", i.e., actions that minimize the validity treat. The tactics adopted in a systematic literature review are those suggested by the approach developed by Kitchenham [37] and implemented in this review. Wherever we deviated from the approach of Kitchenham, we mentioned this explicitly in the related section.

5.3. Construct validity

Construct validity is related to generalization of the result to the concept or theory behind the study execution [70]. In our case it is related to potentially subjective analysis.

According to Kitchenham's guidelines [37], data extraction should be performed independently by two or more researchers, which was not the case in our study. On the other hand, not all relevant publications contained all the relevant data, and where the behavior of the system was evident, even if it was not mentioned explicitly, we used our assessment. Furthermore, during the synthesis phase, we normalized the terms describing the similar phenomena using generic terms, from which we also built a taxonomy.

We minimized these validity threats by asking the authors of publications to confirm the extracted and normalized data. We received replies related to ten RSSEs. We briefly summarize their reactions below.

CloneDetective [34] is useful throughout the implementation and maintenance phases, and not only for a clone removal, as we presumed; the base output form is a document with a report, but also dashboards are supported. Moreover, we overlooked that models, natural language documents, test cases, and XML files are part of the input used by the system. The authors of [10] informed us that their system is reactive. At [56] we modified problems that software engineers are facing from "software engineer doesn't have enough knowledge" to "difficult to search due to big data sets"; we also added "associated Javadoc for each recommended method, words most likely to be relevant in the recommended methods and Javadoc, and whether the software engineer has visited the recommended method in the past" to information and recommendation types in the output. Rascal [45] does not support only code reuse actions, but an entire implementation phase; we also had to remove "current practice is expensive" from potential problems addressed by the system.

The sample contained ten tools from which we extracted eight data types[12]. Out of 80 fields eight were wrong. The average validity of the sample is 90% and the sample's standard deviation is 0.3. The sample was quasi-random, based on a voluntary participation of authors. Due to its size we could use the rules of a normal distribution, even though we do not know the variance of the population. With the 95% confidence, we can say that the extracted data validity is not below 85%.

We later merged the extracted and normalized data into bigger groups. Since the authors of the papers did not review the merge, we pair-reviewed the taxonomy and the merges, to satisfy Kitchenham's requirement.

[12]In this set we did not include the basic data, i.e., publication title, authors, year, and name of the tool.

5.4. External validity

External validity threats are related to the ability to generalize the results of the experiment to industrial practice [70]. In the case of a literature review, the external validity depends on the identified literature: if the identified literature is not externally valid, neither is the synthesis of its content. By the choice of our exclusion criteria, we excluded papers that did not already demonstrate a practical implementation of the presented idea. This tactics supports external validity.

6. Conclusions

When it comes to the outputs of RSSEs, the authors rely on relatively similar types of visualizations, i.e., most of the tools are using simple lists and there are others which are using tables or charts. We believe that investing more research effort in analyzing different and novel user interfaces could be beneficial. Furthermore, it is necessary that researchers consider providing more sophisticated recommendations than most of the systems included in this review provide at the moment, i.e., source code artifacts or other raw data, e.g., other project documents or names of collaborators. It seems that most of the authors of the tools are satisfied with only displaying certain artifacts and expecting that the developer will figure out by herself what to do with this information. That could be one of the main reasons why RSSE systems developed by the scientific community are almost non-existent in practice. We strongly encourage the researchers to put more effort into building RSSE systems that can explain recommendations and (to some extent) tell developers what to do.

In our opinion, largely unexploited opportunities lie in other kinds of potentially relevant information, for example, process related information, which could help developers to comply with team or enterprise best practices, or improve their own development process. We are already investigating some of them, namely, we are analyzing individual perceptions of software engineers to discover how they perceive the usefulness of visualizations of unfinished tasks, effort spent per artifact, last software engineers that were engaged in a development of a module, etc. [25].

Moreover, there is no comprehensive support for the testing phase: we only detected one tool, which supports reuse of test steps. Since software testing is a popular research field at the moment, we expect that in the future there will be a migration of theoretical findings into practice. Certainly the tool support will be required and we think that a promising platform for a new kind of RSSE system is on a horizon.

On the other hand, debugging, maintenance, and implementation phases and reuse actions are well-exploited research topics in the domain of RSSE systems. We also identified some approaches to support a learning process, search for help or an expert, and merging of branches in a revision control.

The identified RSSEs are very task specific, but rarely take into account broader context that could improve the behaviour of the system. For example, we identified some tools that are targeting dispersed development teams or newcomers, but none of them is using an enough sophisticated user model that would allow the adaptation of recommendation visualizations to a specific user, for example; on the other hand, advanced user models are indispensable in recommender systems applied in other domains, e.g., e-commerce. Furthermore, the RSSE systems take into account very limited contextual information related to the specific situation. For instance, some systems adapt the recommendations based on the source code artifact under development, but none is considering current IDE environment.

To generate more relevant recommendations we suggest extensive inclusion of context information, since it improved the quality of recommendations already in other RS domains [1]. None of the systems included in this review takes into account broader context, which would include different types of information, such as information about the environment, timing, user, etc. In addition to more targeted and personalized recommendations, such information could also enable detection of time intervals when the developer can be interrupted with proactive recommendations or it could enable creation of new types of RSSEs.

Currently, we are analyzing which context factors may influence perceived usefulness and perceived ease of use of IDE command recommendations. In our initial research, Gasparic and Ricci [26] identified four types of context: current practice, environment, interaction, and recommendation presentation context.

If it is still too early for the inclusion of *advanced* context models in RSSE systems, we encourage researchers to at least consider the development of context specific RSSEs. For example, the development for smart phones is increasingly popular, but there are no RSSE tools that would support the optimization of source code for energy consumption.

Nevertheless, even if the new RSSE systems will start to collect additional data, we think that the authors of RSSEs should relieve software engineers of manual effort, even when the required effort is low. For example, RSSEs expecting a submission of a query can deduct it from the context and request a confirmation or propose a generated query as a default value, instead of requiring manual input.

Furthermore, we believe that proactive recommending has much greater potential than reactive, even though reactive RSSEs are more common than proactive, since software engineers are usually unaware that they are wrong, when they are; and in such a case, they will probably not consult a reactive system. Proactive RSSEs can instantly redirect software engineers from a wrong path, but they should not distract them. We hope that the development of proactive RSSEs was not completely discouraged by the failure of the infamous Clippy, which was not only due to the proactive behaviour [69].

This SLR provides information about existing RSSE tools and implemented approaches that were presented in scientific journals and conferences, and reveals some research gaps still present in the RSSE field. It can be used as a starting point for new researchers, as well as a motivation for new developments in this increasingly popular research field.

[1] Adomavicius, G., Mobasher, B., Ricci, F., Tuzhilin, A., 2011. Context-aware recommender systems. AI Magazine, 67–80.

[2] Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I., 2013. Software architecture optimization methods: A systematic literature review. IEEE Transactions on Software Engineering, 658–683.

[3] Ali, S., Briand, L. C., Hemmati, H., Panesar-Walawege, R. K., 2010. A systematic review of the application and empirical investigation of search-based test case generation. IEEE Transactions on Software Engineering, 742–762.

[4] Andric, M., Hall, W., Carr, L., 2004. Assisting artifact retrieval in software engineering projects. In: ACM symposium on Document engineering.

[5] Antunes, B., Cordeiro, J., Gomes, P., 2012. An approach to context-based recommendation in software development. In: ACM conference on Recommender systems. pp. 171–178.

[6] Antunes, B., Cordeiro, J., Gomes, P., 2012. Sdic: Context-based retrieval in eclipse. In: International Conference on Software Engineering.

[7] Arcoverde, R., Macia, I., Garcia, A., Von Staa, A., 2012. Automatically detecting architecturally-relevant code anomalies. In: International Workshop on Recommendation Systems for Software Engineering.

[8] Ashok, B., Joy, J., Liang, H., Rajamani, S. K., Srinivasa, G., Vangala, V., 2009. Debugadvisor: a recommender system for debugging. In: Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering.

[9] Bacchelli, A., Ponzanelli, L., Lanza, M., 2012. Harnessing stack overflow for the ide. In: International Workshop on Recommendation Systems for Software Engineering.

[10] Barbosa, E., Garcia, A., Mezini, M., 2012. A recommendation system for exception handling code. In: International Workshop on Exception Handling.

[11] Basili, V. R., Caldiera, G., Rombach, H. D., 1994. The Goal Question Metric Approach. John Wiley & Sons.

[12] Basili, V. R., Trendowicz, A., Kowalczyk, M., Heidrich, J., Seaman, C., Münch, J., Rombach, D., 2014. Aligning Organizations Through Measurement: The GQM+Strategies Approach. The Fraunhofer IESE Series on Software and Systems Engineering. Springer International Publishing.

[13] Beck, K., Andres, C., 2004. Extreme Programming Explained: Embrace Change. Addison-Wesley.

[14] Canfora, G., Di Penta, M., Oliveto, R., Panichella, S., 2012. Who is going to mentor newcomers in open source projects? In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering.

[15] Christidis, K., Paraskevopoulos, F., Panagiotou, D., Mentzas, G., 2012. Combining activity metrics and contribution topics for software recommendations. In: International Workshop on Recommendation Systems for Software Engineering.

[16] Cordeiro, J., Antunes, B., Gomes, P., 2012. Context-based recommendation to support problem solving in software development. In: International Workshop on Recommendation Systems for Software Engineering.

[17] Dagenais, B., Robillard, M. P., 2008. Recommending adaptive changes for framework evolution. In: International Conference on Software Engineering.

[18] Danylenko, A., Lowe, W., 2012. Context-aware recommender systems for non-functional requirements. In: International Workshop on Recommendation Systems for Software Engineering.

[19] Denninger, O., 2012. Recommending relevant code artifacts for change requests using multiple predictors. In: International Workshop on Recommendation Systems for Software Engineering.

[20] Dieste, O., Juristo, N., 2011. Systematic review and aggregation of empirical studies on elicitation techniques. IEEE Transactions on Software Engineering, 283–304.

[21] Dotzler, G., Veldema, R., Philippsen, M., 2012. Annotation support for generic patches. In: International Workshop on Recommendation Systems for Software Engineering.

[22] Elberzhager, F., Kremer, S., Munch, J., Assmann, D., 2012. Guiding testing activities by predicting defect-prone parts using product and inspection metrics. In: EUROMICRO Conference on Software Engineering and Advanced Applications.

[23] Fenton, N. E., Pfleeger, S. L., 1998. Software Metrics: A Rigorous and Practical Approach. PWS Publishing.

[24] Gall, H., Fluri, B., Pinzger, M., 2009. Change analysis with evolizer and changedistiller. IEEE Software, 26–33.

[25] Gasparic, M., Janes, A., Hericko, M., Succi, G., 2013. Metrics-based recommendation system for software engineering. In: Collaboration, Software and Sercives in Information Society.

[26] Gasparic, M., Ricci, F., 2015. Modeling context-aware command recommendation and acceptance in an IDE. In: International Workshop on Context for Software Development.

[27] Ghanam, Y., Maurer, F., Abrahamsson, P., 2012. Making the leap to a software platform strategy: Issues and challenges. Information and Software Technology, 968–984.

[28] Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering, 1276–1304.

[29] Happel, H.-J., Maalej, W., 2008. Potentials and challenges of recommendation systems for software development. In: International Workshop on Recommendation Systems for Software Engineering.

[30] Holmes, R., 2009. Do developers search for source code examples using multiple facts? In: Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation.

[31] Holmes, R., Ratchford, T., Robillard, M., Walker, R., 2009. Automatically recommending triage decisions for pragmatic reuse tasks. In: IEEE/ACM International Conference on Automated Software Engineering.

[32] Holmes, R., Walker, R., 2010. Customized awareness: recommending relevant external change events. In: International Conference on Software Engineering.

[33] Ichii, M., Hayase, Y., Yokomori, R., Yamamoto, T., Inoue, K., 2009. Software component recommendation using collaborative filtering. In: Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation.

[34] Juergens, E., Deissenboeck, F., Hummel, B., 2009. Clonedetective - a workbench for clone detection research. In: International Conference on Software Engineering.

[35] Kawrykow, D., Robillard, M., 2009. Improving api usage through automatic detection of redundant code. In: IEEE/ACM International Conference on Automated Software Engineering.

[36] Kim, J., Lee, S., Hwang, S.-W., Kim, S., 2013. Enriching documents with examples: A corpus mining approach. ACM Transactions on Information Systems, 1–27.

[37] Kitchenham, B., 2004. Procedures for performing systematic reviews. Keele, UK, Keele University.

[38] Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering. Tech. rep., Software Engineering Group, School of Computer Science and Mathematics, Keele University.

[39] Kitchenham, B., Mendes, E., Travassos, G. H., 2007. Cross versus within-company cost estimation studies: A systematic review. IEEE Transactions on Software Engineering, 316–329.

[40] Kobayashi, T., Kato, N., Agusa, K., 2012. Interaction histories mining for software change guide. In: International Workshop on Recommendation Systems for Software Engineering.

[41] Landhausser, M., Genaid, A., 2012. Connecting user stories and code for test development. In: International Workshop on Recommendation Systems for Software Engineering.

[42] Lopez, N., van der Hoek, A., 2011. The code orb: supporting contextualized coding via at-a-glance views (nier track). In: International Conference on Software Engineering.

[43] Lozano, A., Kellens, A., Mens, K., 2011. Mendel: Source code recommendation based on a genetic metaphor. In: IEEE/ACM International Conference on Automated Software Engineering.

[44] Malheiros, Y., Moraes, A., Trindade, C., Meira, S., 2012. A source code recommender system to support newcomers. In: IEEE Annual Computer Software and Applications Conference.

[45] Mccarey, F., Cinnéide, M. O., Kushmerick, N., 2005. Rascal: A recommender agent for agile reuse. Artificial Intelligence Review, 253–276.

[46] McMillan, C., Hariri, N., Poshyvanyk, D., Cleland-Huang, J., 2012. Recommending source code for use in rapid software prototypes. In: International Conference on Software Engineering.

[47] Moraes, A., Silva, E., da Trindade, C., Barbosa, Y., Meira, S., 2010. Recommending experts using communication history. In: International Workshop on Recommendation Systems for Software Engineering.

[48] Murakami, N., Masuhara, H., 2012. Optimizing a search-based code recommendation system. In: International Workshop on Recommendation Systems for Software Engineering.

[49] Muslu, K., Brun, Y., Holmes, R., Ernst, M., Notkin, D., 2012. Improving ide recommendations by considering global implications of existing recommendations. In: International Conference on Software Engineering.

[50] Niazi, M., Ikram, N., Bano, M., Imtiaz, S., Khan, S., 2013. Establishing trust in offshore software outsourcing relationships: an exploratory study using a systematic literature review. IET Software, 283–293.

[51] Niu, N., Yang, F., Cheng, J.-R., Reddivari, S., 2012. A cost-benefit approach to recommending conflict resolution for parallel software development. In: International Workshop on Recommendation Systems for Software Engineering.

[52] Niu, N., Yang, F., Cheng, J.-R., Reddivari, S., 2013. Conflict resolution support for parallel software development. IET Software, 1–11.

[53] Palma, F., Farzin, H., Gueheneuc, Y., Moha, N., 2012. Recommendation system for design patterns in software development: An dpr overview. In: International Workshop on Recommendation Systems for Software Engineering.

[54] Pavón, J., Gómez-Sanz, J., 2003. Agent oriented software engineering with ingenias. In: Central and Eastern European conference on Multi-agent systems.

[55] Petticrew, M., Roberts, H., 2005. Systematic Reviews in the Social Sciences: A Practical Guide. Blackwell Publishing.

[56] Piorkowski, D., Fleming, S. D., Scaffidi, C., Bogart, C., Burnett, M., John, B. E., Bellamy, R. K. E., Swart, C., 2012. Reactive information foraging: an empirical investigation of theory-based recommender systems for programmers. In: SIGCHI Conference on Human Factors in Computing Systems.

[57] Robillard, M., Walker, R., Zimmermann, T., 2010. Recommendation systems for software engineering. IEEE Software, 80–86.

[58] Robillard, M. P., Maalej, W., Walker, R. J., Zimmermann, T., 2014. Recommendation Systems in Software Engineering. Springer Berlin Heidelberg.

[59] Rodríguez-González, A., Lagares, A., Lagares, M., Gomez-Berbis, J., Alor-Hernandez, G., Cortes-Robles, G., 2010. Recometh: Using cbr and characteristic weights to recommend a software development methodology in software engineering. In: IEEE International Conference on Software Engineering and Service Sciences.

[60] Salleh, N., Mendes, E., Grundy, J., 2011. Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review. IEEE Transactions on Software Engineering, 509–525.

[61] Schuler, D., Zimmermann, T., 2008. Mining usage expertise from version archives. In: International Working Conference on Mining Software Repositories.

[62] Shimada, R., Hayase, Y., Ichii, M., Matsushita, M., Inoue, K., 2009. A-score: Automatic software component recommendation using coding context. In: International Conference on Software Engineering - Companion Volume.

[63] Unterkalmsteiner, M., Gorschek, T., Islam, A. K. M. M., Cheng, C. K., Permadi, R. B., Feldt, R., 2012. Evaluation and measurement of software process improvement – a systematic literature review. IEEE Transactions on Software Engineering, 398–424.

[64] Čubranić, D., Murphy, G. C., Singer, J., Booth, K. S., 2004. Learning from project history: a case study for software development. In: ACM Conference on Computer Supported Cooperative Work.

[65] Čubranić, D., Murphy, G. C., Singer, J., Booth, K. S., 2005. Hipikat: A project memory for software development. IEEE Transactions on Software Engineering, 446–465.

[66] Venkatesh, V., Morris, M. G., Davis, G. B., Davis, F. D., 2003. User acceptance of information technology: Toward a unified view. MIS Q., 425–478.

[67] Viriyakattiyaporn, P., Murphy, G., 2009. Challenges in the user interface design of an ide tool recommender. In: Workshop on Cooperative and Human Aspects on Software Engineering.

[68] Wang, L., Fang, L., Wang, L., Li, G., Xie, B., Yang, F., 2011. Apiexample: An effective web search based usage example recommendation system for java apis. In: IEEE/ACM International Conference on Automated Software Engineering.

[69] Whitworth, B., 2005. Polite computing. Behaviour & Information Technology, 353–363.

[70] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer.

[71] Zagalsky, A., Barzilay, O., Yehudai, A., 2012. Example overflow: Using social media for code recommendation. In: International Workshop on Recommendation Systems for Software Engineering.

[72] Zhang, C., Yang, J., Zhang, Y., Fan, J., Zhang, X., Zhao, J., Ou, P., 2012. Automatic parameter recommendation for practical api usage. In: International Conference on Software Engineering.