

Potentials and Challenges of Recommendation Systems for Software Development

Hans-Jörg Happel
FZI Forschungszentrum Informatik
Karlsruhe, Germany
happel@fzi.de

Walid Maalej
Technische Universität München
Munich Germany
maalejw@in.tum.de

ABSTRACT

By surveying recommendation systems in software development, we found that existing approaches have been focusing on “you might like what similar developers like” scenarios. However structured artifacts and semantically well-defined development activities bear large potentials for further recommendation scenarios. We introduce a novel “landscape” of software development recommendation systems and line out several scenarios for knowledge sharing and collaboration. Basic challenges are improving context-awareness and particularly addressing information providers.

1. INTRODUCTION

Today’s software developers have to deal with three main challenges. First, they must use diverse technologies and complex frameworks, and thereby ensure a high quality of their work products. Second, they must cope with a huge amount of daily changing information – both inside and outside their projects. Third, they must prove a maximum of productivity, automation and flexibility, in order to manage strict deadlines, limited resources and ever changing work priorities. Questions developers ask themselves several times a day are [8]: Which interface should I use? Is the quality of my code good enough? Who is working on this component? Whom should I notify about my change? Or what should I do next?

As a knowledge- and automation-intensive domain, the idea of supporting software development with recommendation systems to answer developers’ questions is obvious. Various solutions have been proposed, mainly to address the information overload problem by recommending “what similar developers like”. However, while these solutions target information seekers, the role of information providers is not addressed – i.e. information is typically drawn from some preexisting repository. We argue that proactive recommendations should be supportive for both roles: information seekers and information providers.

This paper makes three contributions. We review state-

of-the-art approaches (Section 2), and line out a landscape for software development recommendation systems (Section 3). We then discuss areas of improvements (Section 4) and identify future research directions (Section 5).

2. STATE OF THE ART

In this section we survey recommendation systems in software development, which have been presented in scientific conferences or journals, within the last five years. Selected systems are not only discussed theoretically, but they also provide concrete implementations. We conclude the section with discussing current limitations areas of improvement.

2.1 Surveyed Systems

CodeBroker. CodeBroker [15] aims to foster software reuse by actively recommending methods that are suitable in a context. It consists of a client called “interface agent” and a back-end. The client, implemented in Emacs, queries the back-end and displays suitable results in a special area. The user context includes three parts. One part is the immediate programming task, which is the basis for getting results from the back-end. It is extracted implicitly from the comments and the signature of the method the developer is writing. CodeBroker maintains a “discourse model” that stores methods, which were explicitly invoked by the user. A “user model” captures methods, which the developer already knows and thus does not need to be recommended. Both, the discourse model and the user model are used by the interface agent to filter out results that were returned from the back-end. Recommendations in CodeBroker are triggered continuously as soon as the interface agent is activated.

Dhruv. Dhruv [2] aims to assist the software maintenance process, by recommending relevant information during bug inspection. Therefore, Dhruv is integrated in a web-based bug tracking system and displays recommendations in a special sidebar. Such recommendations may involve source code files, mailing list discussions or similar bug reports. Dhruv does not operate on a special user profile. The context for recommendation is always the bug report, for which related information is retrieved. The recommendation corpus data is created in two steps. First, meta-data is extracted from source code, mailing lists and existing bug reports. Afterwards, algorithms are employed in order to infer relationships among meta-data. Based on the identification of named entities, several heuristics are used to infer the actual

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RSSE '08, November 10, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-228-3 ...\$5.00.

role of the entity in the context of the analyzed artifact. The meta-data as well as possible relations are modeled in ontologies. Thus after analysis, the various meta-data entities form an interconnected semantically described graph structure. Based on this graph structure, recommendations are drawn by relational similarity. The relations in the ontology have weights assigned, which are used for the similarity calculation.

Hipikat. Hipikat [3] strives to support developers (especially newcomers) working on maintenance tasks. It builds a group memory consisting of four types of artifacts: source code, email discussions, change tasks and documents. A developer may use the Hipikat client – an Eclipse plug-in – to query artifacts inside Eclipse for related ones. The Hipikat back-end then returns a list of source code, email discussions or bug reports, which are related to the developers query. Hipikat does not maintain a user profile. Instead, recommendations are based on explicit queries, which must include an artifact reference, for which recommendations are requested. The selection of appropriate results is based on similarity calculations, which operate on the relations between the artifacts. The queried project memory is built automatically from existing artifacts. The relations are inferred by five different, manually implemented heuristics. Examples are a log matcher, which tries to identify bug report IDs in source code documentation using regular expressions, or an activity matcher, which compares check-in times of source code changes with the closing time of bug reports.

Mylyn. While other systems filter relevant source code from large repositories, mylyn [7] targets to optimize the user interface of an Integrated Development Environment (IDE). The core idea is that not all classes in large software projects are relevant for working on a given task. Thus, mylyn identifies and hides or blurs classes which are less relevant. In mylyn developers are sequentially working on limited tasks (e.g. fixing a bug), which affect only a subset of source code files. A task is also the context information for recommendations. During each task, a "degree-of-interest" model is maintained for each source code file. The degree-of-interest is a value, which is influenced by the developers' interaction with the file. This is complemented by a "degree-of-separation" model, which represents relations among the source code files. The combined information yields a value which is interpreted to visually indicate task-related files in the IDE.

RASCAL. The overall approach of RASCAL [10] is similar to the CodeBroker. It is also motivated by the availability of large code repositories, which can not be overseen by developers. The RASCAL system consists of a client for the Eclipse IDE and a server back-end. The client tries to predict the next method the developer would use by analyzing the current class and comparing it to similar classes. Thus, the user context in RASCAL solely consists of information implicitly extracted from the current class a developer is editing. RASCAL extracts the total number of calls to (external) methods inside a class. Following the same principle, the back-end corpus is built by analyzing existing source code. Recommendations in RASCAL must be triggered manually. Upon that, the current class is matched

against the information in the back-end. In a first step, the k-nearest neighbours of the current class are identified. Afterwards, an average order of the methods is created. Finally, those methods are ranked high, which most often occur after the last method in the queries class.

Strathcona. Strathcona [6] is another Eclipse plug-in that aims to recommend source code examples relevant for the current development task. The main application scenario is the usage of third-party libraries. At the back-end, Strathcona extracts facts from given source code and stores it into a relational database. A query has to be triggered by the user by selecting a code fragment. Additionally, Strathcona client automatically extracts a "structural context". This includes the method signature, declaring types (plus super-types), field names, referenced types and fields of the current method. This information is sent to the server, where four different structural similarity heuristics (based on inheritance, method calls, usage and field references) are used to match relevant recommendations. All heuristics are implemented as SQL queries on top of the database. After executing the queries, results from the heuristics are merged and the top 20 are selected and sent back to the client.

2.2 Summary and Areas of Improvement

Table 3 summarizes surveyed systems. A number of limitations could be addressed by future systems.

- Existing systems are limited to either recommend methods to use next or artifacts which are "related" to the current situation.
- Existing systems are based upon a centralized, static corpus. The aspect of information provision is not addressed.
- The description of the user's situation or "context" is limited to single properties such as the current class a user is working in.
- There is no pro-active triggering of information push: recommendations are either triggered automatically in a continuous way, or have to be requested by users.
- Architectures of the surveyed systems are inflexible and do not allow for extensions.

We shortly discuss related challenges in the following.

Architecture. Most of the presented systems use a client/server architectural style and operate on one server exclusively. Thus, the amount of included information is limited by capacity and management effort on the server side. For example, a P2P-based approach makes more information available without introducing performance problems. Furthermore, a decentralized approach makes additional knowledge accessible, which developers would not contribute to a central repository. None of the presented approaches uses a true collaborative filtering approach, which leverages the experience of the developer community. Only mylyn is anticipating knowledge exchange across developers, by allowing exchanging the degree-of-interest model for a given task.

Knowledge Representation. Except Dhruv, all described systems are working with traditional knowledge representations and hard-coded heuristics. A more flexible knowledge representation, e.g. based on Semantic Web technologies, will not only improve the possibilities to integrate and share

Tool	Goal	Architecture	Recommendation	User profile/ Context	Trigger	Corpus	Matching algorithm
Hipikat	Assist newcomers and maintainers	Client/Server (Eclipse)	Project documents Messages/issues	Artifact which is subject of the query (explicit)	Manual query	CVS, Bug reports, E-mails	Relational similarity (content-based)
Code-Broker	Foster source code reuse by suggesting methods	Client/Server (Emacs)	Source code(method)	Current method comments and signature (implicit) Discourse model (explicit) User model (explicit)	Automatic query	JavaDoc and source code	Conceptual similarity (LSI) for comments, and constraint similarity for signature matching (content-based)
Dhruv	Speed up bug fixes by recommending related artifacts	Web application	Code files, Discussions Bug reports	Current bug report (implicit)	Automatically	Community data (code, e-mails, bug reports)	Weighted relational similarity (content-based)
mylyn	Hide non-relevant artifacts for current task	Eclipse plug-in	Source files	Task-based user interaction on files (explicit/implicit)	Automatically	Files in project workspace	Degree of interest, based on clicks (interaction) and class relations (content-based)
RASCAL	Predict next method to be inserted	Client/Server (Eclipse)	Source code (method)	Analysis of current class (implicit)	Manual query	Swing-based applications from SourceForge	Hybrid
Strathcona	Give example code for third-party APIs	Client/Server (Eclipse)	Example code	"Structural context" (Implicit)	Manual query	Source code	SQL-queries (content-based)

Table 1: Overview of software development recommendation systems

additional information, but also help to make system behaviour more transparent – e.g. by providing explanations for recommended items.

Pro-activeness. When compared to traditional recommendation systems, the user profiles created by the described systems are rather simplistic. Thus, recommendations are either triggered automatically in a continuous way, or have to be requested by users. True pro-active assistance should identify certain problem situations (e.g. run time errors, or unexpected program behaviour) based on a richer user context, which allows more focused recommendations.

Automatic Experience Capture. The presented approaches are either focusing on recommending methods to use (Code-Broker, RASCAL, Strathcona) or development artifacts (Dhruv, Hipikat, mylyn). They rely on explicit knowledge, which already exists. In contrast, user observation frameworks can capture problem solving patterns, which are usually not explicitly documented by developers. We consider this kind of information very useful for developers.

3. RECOMMENDATION LANDSCAPE

Based on the survey of existing systems in the previous section, we line out a “landscape” of software development recommendation systems, which considers additional use cases for developer assistance. Therefore we distinguish two major dimensions: the addressed stage of the knowledge sharing process (when to recommend) and the recommended information (what to recommend). Table 2 summarizes the classification.

3.1 When to Recommend

Research indicates that the absence of awareness about the existence of certain knowledge (information access) and the low level of experience sharing and capture (information provision) are two major blockers for knowledge sharing, especially in distributed settings [4]. Given the existence of large amounts of reusable artifacts such as specifications, source code or binaries – in both corporate repositories and the Internet – there is a large potential to improve the efficiency of software development. However, due to constraints

		When	Information Access	Information Provision
		What	Propose...	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 problems
	Quality measures		Problematic change, Patterns to improve quality	How problems has 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

Table 2: Recommendation landscape

in time and mental capacity, it is hard for humans to find information suitable for solving a given problem. Recommendation systems, which provide an intelligent “information push” functionality and suggest information for a given context of user, are thus desirable [3].

Even if large repositories are a good starting point for providing recommendations, we claim that the usefulness of recommendation systems can be improved by considering the role of information providers. Two main arguments support this claim. First, the contribution to central repositories suffers from a number of motivational, organizational and technical barriers [14]. Especially in distributed settings a fragmentation of information can emerge, when developers hoard information locally, even if it might be useful for distant colleagues. Second, particular information which should be recommended depends on an immediate participation of knowledge providers and is per se evolving and present either implicitly in the head of knowledge providers or explicitly on their local working environments. Examples are the rationale behind certain decisions, the steps followed to fix a particular bug or awareness information.

Therefore, we claim that recommendation systems should actively address the role of information providers by encour-

aging users to share certain information with their teams. In collaboration scenarios, a system might recommend two remote developers to, e.g. reveal their current working context if they change the same file. A recommendation system might also ask to share a web page, which a developer extensively used to solve a certain problem.

3.2 What to Recommend

We classify recommended information into development and collaboration information.

3.2.1 Development Information

Code. Recommending code in software development addresses two main problems: the enormous number of building blocks used in software development and the “machine-orientation” of building languages. Developers have to reference libraries, instantiate frameworks and use technologies, which provide evolving interfaces. The Java™2 Platform Standard Edition for example includes 1885 methods that start with an “A”. Syntactic mistakes in combining and using these blocks are not allowed, since the output will be executed by machines. Recommendation systems can assist developers to choose the right building blocks or propose content, which can be used in the current situation. This helps developers to quicker write syntactically correct code and reduces the selection choice of building blocks. The most famous example in this category is code auto-completion in modern IDEs such as Eclipse, Visual Studio or NetBeans. Underlying logic spans from trivial heuristics, e.g. syntactical matching of keyword prefixes, to more advanced ones taking into consideration user annotations, current node in the abstract syntax tree or declared variables and their types.

Artifacts. Software projects include many artifact types such as specifications, design models, source code and test cases. These artifacts are interconnected. Often one artifact includes information which is relevant to understand another one. With the vast number of artifacts whose content and structure changes over time, it is often difficult for developers to have an overview about all needed artifacts for a specific task. Recommendation systems can help to share discovered semantic relations between artifacts as well as used artifacts for specific tasks.

Quality Measures. Recommendation systems can play the role of a “peer reviewer” to continuously check changes and suggest quality improvement measures. Two main features can be offered: detecting and highlighting areas which are error-prone, and recommending patterns to increase the quality of the work product.¹ For example, constructive criticism on the quality of UML designs, with suggestions on problematic design features can be given based on best and worst practices or model checking. Also incremental compilers provides visual clues for compilation errors and warnings, as well as suggestions how to how to fix these issues. More advanced static content analysis can be used to detect bugs from changes or solution patterns used to solve particular bugs. In the latter case developers are asked to share their experiences with others, who will be provided with this in-

¹see e.g. <http://www.intooitus.com/inCode.html>

formation in similar problem situations.

Tools. Sheperd and Murphy claim that only 20% of application features are used by each developer [13]. Using inappropriate features to complete tasks might cause developers to spend more time. Developers who are not aware of a refactoring tool, will spend, e.g. much more time in implementing delegations instead of using refactoring tools to generate them. Recommendation system might use activity logs to deduce questions developers ask, and then coach them automatically on appropriate, possibly unfamiliar tools or features to answer those questions more efficiently.

3.2.2 Collaboration Information

People. As software evolves over time in both design and functionality, the identification of expertise for particular design decisions or particular features becomes an important issue. Especially in agile or distributed projects many information is either not documented or hardly accessible for all developers. Thus, experts – i.e. specialists who know the system very well – are considered an important source of information [1, 12]. Recommendation systems can propose to contact people for particular issues. They can also recommend to share particular personal opinions and experiences to assess the expertise of people in particular fields.

Awareness. In distributed development scenarios, awareness information is of particular importance. A simple scenario like mutually informing two developers who are concurrently modifying the same file is not well supported in current software development tools [11]. In offshore development, remote managers have difficulties to oversee what developers are currently doing. Since such issues bear complicated privacy implications, recommendation systems can assist developers to selectively push such status information when required. In a larger scope, developers are seldom aware of other developers reading, using and modifying the code they have written. Recommendations to better document functionality or design decisions could help to improve reuse and maintenance of such code.

Status and Priorities. Many software projects fail because status information and open issues are not communicated properly and on time. Recommendation system can play a major role in suggesting priority changes and showing personal performance overviews, as well as how others are performing to achieve similar tasks (anonymously). Prioritization is a non-trivial task in a multiple-project setting. Recommendation systems can filter and aggregate parameters to propose new task priorities. Observing explicit priority changes on “key” developers, recommendation systems can also ask to share these priorities or decision rationales to use them in similar situations. Support for such scenarios could significantly improve project and risk management in larger projects.

4. PATHS FOR REALIZATION

In this section we briefly sketch our concept for addressing some of discussed limitations of current recommendation systems. Basic building blocks are improved context-awareness and particularly addressing information providers

by what we call “inverse search”.

Improved context-awareness aims to develop a better understanding of developers activities. Existing work such as mylyn leverages interaction data of developers within the IDE. However, collected data includes low-level, log-like information, and does not allow for much meaningful deductions about developer’s information needs or information provision capabilities. Information such as error messages or developers’ search queries is not collected. Therefore a context observation framework is needed, which aggregates and semantically enriches low-level interactions with the IDE or other working environments (such as the web browser) to more meaningful, human-readable information. In [9] we describe an architecture for such a framework, which we are realizing within the Open Source platform TeamWeaver².

Collecting context information raises questions about privacy and control. We believe that researchers on recommendation systems have to give more attention to privacy issues. We believe privacy can be protected by keeping context information private by default, but pro-actively recommending to share information if it would be helpful for other team members. The underlying concept of inverse search [5] maintains a private information need model for each developer. Parts of this model can be anonymously shared on a server, which aggregates individual information needs to a more general team information need. This information is offered as a service and can be retrieved by other developers who might be able to provide information which can help satisfying existing needs. A recommendation system could indicate such opportunities for contributing to a development teams’ information space.

The combination of context-awareness and assistance on information provision addresses several important issues of current approaches. It allows access to content and artifacts in the private space of developers, without threatening their privacy. In concert with the automated capture of context information, this can also be used to realize scenarios for which data is currently not available. Also, increased context-awareness could make recommendations more specific – e.g. by autonomously providing assistance when development problems are identified.

5. CONCLUSIONS

Due to the large amount of artifacts in software development projects, recommendations systems have become popular to assist developers in finding reusable and related content. However, our analysis shows that current systems have a number of limitations such as their non-flexible architecture and the negligence of implicit context information. We identified the following research areas:

- Recommendations to capture experiences and share information.
- Semantic analysis and description of working context.
- Automatic context-aware triggering of recommendations.

6. REFERENCES

- [1] Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from cvs. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 125–128, New York, NY, USA, 2008. ACM.
- [2] Anupriya Ankolekar, Katia Sycara, James Herbsleb, Robert Kraut, and Chris Welty. Supporting online problem-solving communities with the semantic web. In *WWW '06*, New York, NY, USA, 2006. ACM.
- [3] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, 2005.
- [4] Kevin C. Desouza and J. Roberto Evaristo. Managing knowledge in distributed projects. *Commun. ACM*, 47(4):87–91, 2004.
- [5] Hans-Jörg Happel. Closing information gaps with inverse search. In *7th International Conference on Practical Aspects of Knowledge Management*, Lecture Notes in Computer Science. Springer, 2008.
- [6] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.
- [7] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.
- [8] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *ICSE'07*, 2007.
- [9] Walid Maalej and Hans-Jörg Happel. A lightweight approach for knowledge sharing in distributed software teams. In *7th International Conference on Practical Aspects of Knowledge Management*, Lecture Notes in Computer Science. Springer, 2008.
- [10] Frank Mccarey, Mel Ó. Cinnéide, and Nicholas Kushmerick. Rascal: A recommender agent for agile reuse. *Artif. Intell. Rev.*, 24(3-4):253–276, 2005.
- [11] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantir: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.
- [12] David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM.
- [13] David C. Shepherd and Gail C. Murphy. A sketch of the programmer’s coach: making programmers more effective. In *CHASE '08: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, pages 97–100, New York, NY, USA, 2008. ACM.
- [14] Cynthia T. Small and Andrew P. Sage. Knowledge management and knowledge sharing: A review. *Information, Knowledge, Systems Management*, 5(3):153–169, 2006.
- [15] Yunwen Ye and Gerhard Fischer. *Automated Software Engineering*, 12(2):199–235, 2005.

²<http://www.teamweaver.org>