This article has been accepted
Some content may change

Recommendation Systems for Software Engineering

Martin P. Robillard, School of Computer Science, McGill University, Montréal, QC, Canada
martin@cs.mcgill.ca

Robert J. Walker, Department of Computer Science, University of Calgary, Calgary, AB, Canada
walker@ucalgary.ca

Thomas Zimmermann, Microsoft Research, Redmond, WA, USA
tzimmer@microsoft.com

Abstract

Software development can be challenging because of the large information spaces that developers must navigate. Without assistance, developers can become bogged down, and spend a disproportionate amount of their time seeking information at the expense of other value-producing tasks. Recommendation Systems for Software Engineering are software tools that can assist developers with a wide range of activities, from reusing code to writing effective bug reports. We provide an overview of recommendation systems for software engineering: what they are, what they can do for developers, and what they might do in the future.

Keywords

D.2.6 Programming Environments/Construction Tools

D.2 Software Engineering

D.2.3 Coding Tools and Techniques

D.2.2 Design Tools and Techniques

Digital Object Identifier 10
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

Despite steady advancement in the state of the art, software development remains a challenge. We are continually introduced to new technologies, new components, and new ideas we can draw from. The systems we work on have more code and depend on larger libraries. Mastering a programming language is no longer sufficient to ensure software development proficiency: developers must also learn to navigate large code bases and class libraries. For example, a developer with a task as mundane as adding a message to a status bar may have to discover the right classes among thousands in a class library, and then understand complex interactions between them. Without assistance, developers can become bogged down in a morass of details, and may need to spend a disproportionate amount of their time seeking information at the expense of other value-producing tasks.

In recent years, various kinds of *recommendation systems* have emerged to help users find information or make decisions in cases where they lack experience or cannot consider all the data at hand [1]. Recommendation systems combine "ideas and techniques from information filtering, user modeling, artificial intelligence, user interface design and human-computer interaction [to] provide users with proactive suggestions that are tailored to meet their particular information needs and preferences" [6]. To date, recommendation systems have mostly been tied to the web, providing features that range from recommending books or movies to helping people find news items that may be relevant to them. Many web-available recommendation systems embody mature technology delivered as part of commercial systems (for example, the recommendations on Amazon.com). The challenges faced by people navigating large information spaces are, in some ways, similar to the challenges faced by software developers. Consider the parallel between a traveler trying to choose between over 300 hotels in New York and a developer trying to find the one class that suits their needs among the 300 in a class library. To assist developers with software development tasks, a number of recommendation systems for software engineering are starting to emerge, a trend that is more recent than recommendation systems for the web. Recommendation systems in the software engineering domain can assist developers with a wide range of activities, from reusing code to writing effective bug reports.

Why are recommendation systems for software engineering emerging now? A variety of factors have recently interplayed to promote and facilitate their development. These include:

- *The increasing scale of systems, available libraries, and frameworks* leads to increased risks that developers will become overwhelmed with information, even when they are familiar with the system under development.
- *An ever-quickening pace of software evolution* means that existing knowledge about a system (e.g., as captured in documentation) is quickly invalidated, and must be acquired afresh, hence increasing the need to navigate large information spaces frequently.
- *A trend towards technologically heterogeneous systems* means that developers will often need to acquire information outside of their specialization.
- *An increase in distributed development* imposes constraints on the availability of team members to answer questions and share their knowledge, making technological alternatives necessary.

In addition, key enabling factors have given rise to practical recommendation systems for software engineering: large masses of publicly available source code that can be analyzed for recommendations,

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

mature software repository mining techniques, and the mainstream adoption of common interfaces for software development (with web interfaces such as Bugzilla or tool integration platforms like Eclipse).

Recommendation systems for software engineering are ready to make their entrance into industrial software developers' toolboxes: research prototypes are quickly gaining in maturity, tools are being released, and first-generation systems are being re-implemented in different environments. A workshop we recently organized also showcased many of the future directions for recommendation systems for software engineering, including an ever-widening array of supported tasks [10]. In this article, we provide an overview of recommendation systems for software engineering: what they are, what they can do for developers, and what they might do in the near future.

What Are Recommendation Systems for Software Engineering?

We describe RSSEs by starting with a general definition and description of recommendation systems as proposed by the organizers of the *ACM International Conference on Recommender Systems*:

> [Recommendation] systems are software applications that aim to support users in their decision-making while interacting with large information spaces. They recommend items of interest to users based on preferences they have expressed, either explicitly or implicitly. The ever-expanding volume and increasing complexity of information […] has therefore made such systems essential tools for users in a variety of information seeking […] activities. [Recommendation] systems help overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance. [1]

In the context of software engineering, recommendation systems support developers in their decision-making, but particularly with their information finding goals. Examples of information-finding activities supported by RSSEs include: finding the right code, application programming interface (API), and developer. In the case of software engineering, the "large information spaces" of interest are a subset of a system's code base, its libraries, bug reports, version history, and other documentation.

The above quote also highlights the relation between the output of recommendation systems and an expression of their users' interests. Developers can express their interest explicitly (for example, by requesting a recommendation through a query, by marking elements as interesting), and implicitly (for example, by having their actions monitored and the recommendations issued automatically at the opportune time). This distinction highlights a key challenge for recommendation systems in general: how to establish the context used to make a recommendation. The context is potentially all the information about the user, their working environment, and their work, that are available at the time of the recommendation, and which can establish the relevance of the recommendation. In recommendation systems for traditional domains, the context is typically established through a user profile, which can be composed of any combination of user-specified and learned characteristics. For example, when Netflix recommends movies, it considers the genre preferences and movie ratings by the customer (user-specified), but also the movies they have rented in the past (learned).

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

In recommendation systems for software engineering, it can be very challenging to establish the context, because of the comparatively rich range of activities associated with tasks supported by RSSEs. A traveler who seeks a hotel recommendation can typically specify much of the context with a handful of simple criteria (such as price, services, star-rating, distance from area of interest). In contrast, to provide a software developer with a recommendation about where to look next when exploring the source code, the recommendation system must establish a number of rather fuzzy parameters, such as what the developer is interested in discovering, what the developer already knows, and which parts of the source code are related to the developer's needs.

In the software engineering domain, the following aspects could all be considered as part of the context that may need to be provided to or inferred by a recommendation system:

- the characteristics of the user (e.g., job description, expertise level, prior work, social network),
- the kind of task being conducted (e.g., adding new features, debugging, optimizing),
- the specific characteristics of the task being conducted (e.g., code edited, code viewed, dependencies), and
- the past actions of the user or the user's peers (e.g., which artifacts have been viewed, which artifacts have been explicitly recommended).

The last part of the definition for *general* recommendation systems underlines the qualities of the output of a recommendation system: "novelty, surprise, [or] relevance". If recommendation systems are to assist developers, they must provide them with valuable information; information is not usually valuable if developers already know it, or if it is not relevant to their problem. This aspect is another challenge faced by RSSEs because the value of the recommendations they issue is subjective and situation-specific. Sometimes a recommendation is valuable when the developer wasn't even aware of a need, e.g., "I didn't realize that making this change would be risky"; sometimes a recommendation is valuable if it merely corroborates the developer's suspicions, e.g., "I *think* that this is the only dependency". Considering all these particulars of the software engineering domain, we define RSSEs as follows.

> **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.*

What Can RSSEs Do for Developers?

RSSEs can help developers to discover information that they should know about and to evaluate alternatives when making decisions. Both of these types of activities span a wide spectrum of software engineering tasks and involve considering practically unbounded amounts of software development data; thus, such activities can benefit from automated assistance.

The majority of RSSEs support developers while *programming*. Current RSSEs have demonstrated potential for surfacing opportunities for reuse (CodeBroker [13]) and for locating experts to consult (Expertise Browser [7]). RSSEs also facilitate deciding what examples to use (Strathcona [5]) and what

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

sequence of calls to make (PARSEWeb [12]). RSSEs can support developers when *navigating* large code bases by boiling down rich and complex information spaces into clearly prioritized lists of alternatives: where to look in the code (Suade [9]) and what to change next (eROSE [14]).

However, RSSEs also support software engineering activities besides programming. In the context of *maintenance*, RSSEs can recommend what methods to use to adapt code to a new version of a library (SemDiff [3]). When *debugging*, RSSEs can help to find code and people related to a bug fix (Dhruv [2]). For *testing*, RSSEs can make predictions about which parts of a software product will have the most defects and thus help to prioritize resources for quality assurance [8].

A Sample of RSSEs

Although the diversity of RSSEs that have been developed makes generalizations about their architecture difficult, most RSSEs involve at least three main pieces of functionality. A *data collection mechanism* collects software development artifacts and data that are the byproducts of the software development process into a model that can be used to produce recommendations. A *recommendation engine* analyses the data model to produce recommendations. Recommendations are then presented in the RSSE's *user interface*, which can also be used to trigger the recommendation cycle.

A good way to understand more concretely what RSSEs look like and what they can do for developers is to look at a few examples. We present three examples of recommendation systems that we have developed and experimented with. This small sample does not represent the entire field of RSSEs: aside from our deep understanding of their implementation, we selected these three systems to illustrate how recommendation systems work, and how they can differ in key design dimensions.

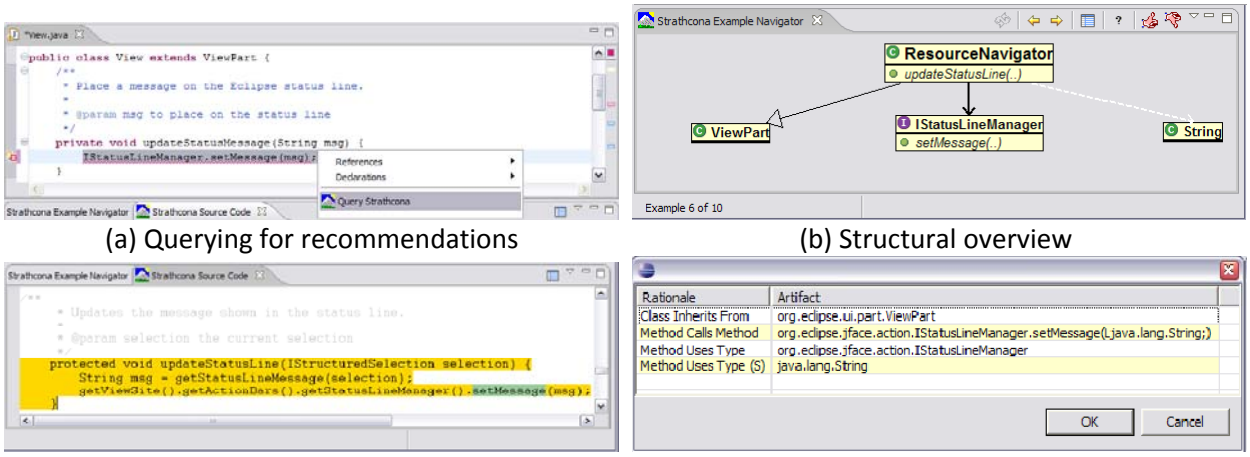
Guiding Software Changes with eROSE. If you browse books at Amazon, you may have encountered recommendations like "Customers who bought this book also bought…" Such findings stem from Amazon's purchase history: buying two or more books together establishes a relationship between books, which is used by Amazon to create recommendations. The eROSE plug-in for Eclipse realizes a similar feature for software development by mining past changes [14] from version archives like CVS. When used by a developer, it keeps track of the elements changed (the context) and constantly updates its recommendations in a view after every save operation. For example, when a developer wants to add a new preference to the Eclipse IDE and has changed `fKeys[]` and `initDefaults()`, eROSE would recommend "Change `plugin.properties`" because all developers who changed the code of Eclipse did so in the past. During setup eROSE preprocesses the project's CVS archive to identify fine-grained changes to program elements such as classes, methods, and fields. In addition, eROSE groups elements that were changed at the same time and by the same developer ("co-change") into transactions and stores them in a SQL database. The context is then used to query the set of transactions. Transactions that contain at least one element of the context are retrieved, combined, and returned. From the result, eROSE derives its recommendations: First it excludes the context because the user has changed these elements already. Second it ranks the remaining elements by the number of transactions they belong to; the more frequent an element, the more likely the user should change it. Because the underlying concept of eROSE is co-change, it is fairly language independent and can recommend text, image, or

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

documentation files in addition to program elements. Furthermore, eROSE can reveal hidden dependencies such as whenever the developer changes code to create a database, the PNG file depicting the database schema has to be updated. A prototype implementation of eROSE is available at http://www.st.cs.uni-saarland.de/softevo/erose/.

Finding Relevant Examples with Strathcona. Frameworks provide developers with a repository of code that can help them in their coding tasks. However, frameworks are frequently large and difficult to understand; documentation is often incomplete or insufficient to help developers with the specific tasks they are working on. The Strathcona system [5] automatically provides developers with relevant source code examples to help them in using frameworks properly and effectively. For example, when a developer becomes stuck while trying to figure out how to change the status bar in the Eclipse IDE, they can highlight their partially-complete code (the context) and ask Strathcona for similar examples. A set of structural facts are extracted from the code fragment, like what types are referenced (`IStatusLineManager`, an abstract interface type) and what methods are called (`setMessage(String)`). Strathcona searches for occurrences of each of these facts in a code repository, via PostgreSQL queries. Strathcona then uses a set of heuristics to decide on which examples are the best. Examples that are selected by the most heuristics are ordered before those chosen by fewer heuristics; the top 10 results are returned. It presents its results both with a structural overview diagram and with source code highlighted to show similarities to the developer's partially-complete code; a rationale can also be viewed to see why the example has been proposed (see Figure 1). A prototype and more details are available at http://lsmr.cs.ucalgary.ca/strathcona.



(a) Querying for recommendations (b) Structural overview

(c) Highlighted source view (d) Rationale view

Figure 1: Querying for recommendations, and the resulting presentation of examples and rationales in Strathcona.

Guiding Software Navigation with Suade. Suade is an Eclipse plug-in that automatically generates suggestions for software investigation [8]. When a developer is exploring the code to complete a change task and becomes stuck, not knowing where to look next among all the elements related to the current task, they can use Suade to trigger recommendations about where to look next. Based on a set of fields and methods explicitly specified as "relevant" by a developer (the context), Suade automatically

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

retrieves all the elements that are directly related to the elements in the context through method call and field access relations, and ranks them based on their likely relevance to the investigation in a context-specific manner. Suade ranks the recommended fields and methods by extracting all the static dependencies to context elements from the project's source code into a dependency graph, and by analyzing the topology of this dependency graph with heuristics. For example, if a method in the source code only calls methods that a developer specified as relevant, it is ranked more highly than methods that call the context methods in addition to many others. In Suade, the user creates a context by dragging and dropping elements of interest into a view. Whenever a context is specified, a developer can trigger a recommendation cycle. Suade displays recommendations as items in a list, in a dedicated view. Suade supports iterative recommendation generation by allowing users to drag recommended elements back into the context view, to generate updated recommendations. More information on Suade, including screenshots and a downloadable version of the tool, is available at http://www.cs.mcgill.ca/~swevo/suade/.

RSSEs at Large

The examples described in the previous section only represent a small fraction of the spectrum of RSSEs available and in development. We can take a broader look at the field by considering three different design dimensions for RSSEs: nature of the context, recommendation engine, and output modes (see Table 1 for a summary, and the sidebar for descriptions of the additional RSSEs mentioned in this section).

<table>
<tr><th>Nature of the Context</th><th>Recommendation Engine</th><th>Output Mode</th></tr>
<tr><td rowspan="3">INPUT
explicit | implicit | hybrid</td><td>DATA
source | change | bug reports | mailing lists | interaction history | peers' actions</td><td>MODE
push | pull</td></tr>
<tr><td>RANKING
yes | no</td><td>PRESENTATION
batch | inline</td></tr>
<tr><td colspan="2">EXPLANATIONS
from none to detailed</td></tr>
<tr><td colspan="3">USER FEEDBACK
none | locally-adjustable | individually-adaptive | globally-adaptive</td></tr>
</table>

Table 1: Design dimensions for RSSEs

Nature of the context

One of the concepts at the core of RSSEs is that of the context in which a recommendation is provided. The context forms the input to the RSSE, and can be provided explicitly, implicitly, or as a hybrid of these strategies.

Developers can provide context explicitly through various means of interaction with the user interface, including entering text, selecting elements directly in the code (as in Strathcona or PARSEWeb), or dragging and dropping elements into an explicit "context" widget (as in Suade). Specifying the context

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

explicitly is appropriate when supporting tasks where the context is very difficult to detect (e.g., the interest of a user or their level of experience), and where the amount of information to provide is limited. For many tasks, part of the context can be obtained implicitly by the recommendation system. For example, this is the case for RSSEs that track and react to actions by developers (as in eROSE), or for RSSEs that require context information that would be unreasonable to specify explicitly (like the history of interaction between a developer and their IDE). Finally, in many cases a combination of implicit and explicit context-gathering will be necessary. With Strathcona, the developer explicitly selects a section of code text, but the code is parsed and analyzed and a structured model is implicitly extracted to form the context for the recommendation system. In the case of CodeBroker, the context is automatically extracted from code comments , a syntactic construct that is assumed to be part of the task.

Recommendation Engine

In addition to the context, RSSEs need to analyze additional data to provide their recommendations. This data can include the source code of a project, the complete history of changes for a system, other artifacts like emails posted to mailing lists and bug reports, accumulated interaction data from many programming sessions, test coverage reports, and code bases external to the project. Analyzing these data sources is often referred to as *mining software repositories (MSR)*, which was discussed in a recent special issue of IEEE Software (January/February 2009). In the case of RSSEs, mining software repositories is just one potential means to an end. Some RSSEs, such as Suade, do not rely on MSR to produce recommendations.

Every RSSE we have encountered uses a *ranking mechanism* as a cornerstone of its analysis. An ideal ranking algorithm systematically "recommends" the items that are most valuable to the user at the top of its rankings. In practice, providing rankings relies on a model of what a developer will find useful, and such models are never perfect, because they must model not only the task, but also the developer's individual perspective on the task: What's useful for a developer may not be useful for that developer's colleague. Finally, models used by recommendation engines may also have to take into account the time-sensitivity of recommendations: what's useful for a developer now may not have been useful to that developer in the past, or might not be useful in the future.

Output Modes

Most existing RSSEs operate in *pull mode* and produce recommendations after an explicit request from the developer, which can be as simple as a single click in an IDE. Some RSSEs operate in *push mode*, delivering their results continuously (e.g., eROSE, CodeBroker, Dhruv). The danger of push mode is that it can become obstructive if not designed well. Conversely, the danger of pull mode is that developers may miss something important if they don't even think to ask about it.

We can also distinguish a *batch* output mode of use, when a developer wants a complete set of recommendations about a task and thus is willing to go to a separate view in an IDE (the typical approach in existing RSSEs); and an *inline* output mode, when annotations are made atop artifacts that the developer is otherwise perusing (as in Dhruv).

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

Cross-Dimensional Aspects

A number of RSSE features cross design dimensions. The recommendation engine of an RSSE can also take into account the developer's interactions with the RSSE itself, so that bad recommendations can be flagged, and eliminated from future results. In other words, past recommendations support a feedback mechanism and can become part of the context or the data the RSSE operates on. Ranking mechanisms may be: locally-adjustable, in which the developer can adjust the inferred context manually (e.g., Suade); individually-adaptive, wherein the algorithm is refined for the sake of the individual based on their own implicit or explicit feedback (e.g., CodeBroker); or globally-adaptive, where feedback from one user can affect another user. Existing RSSEs tend to be limited in this dimension.

Finally, RSSEs vary in how they can explain their results. On one end, some systems provide recommendations that appear to be almost magical. For example, some systems predict files as defect-prone without providing any explanation, which makes it difficult for developers to trust such recommendations. At the other end, some systems provide detailed rationales justifying each recommendation (e.g., Strathcona). Naturally, providing a detailed rationale for a recommendation requires exposing part of the inner workings of the RSSEs, which has both potential benefits (e.g., to increase confidence in the recommendation) and pitfalls (e.g., information overload).

Limitations and Potential of RSSEs

While RSSEs hold out possibilities for advancing the state of the art in software development tools, they are not without their limitations. For example, when large repositories of information are necessary, RSSEs can be faced with the *cold-start problem* where required information cannot be supplied until a project is underway, although one solution is to leverage analogous data from other projects instead. Additionally, RSSEs cannot crawl inside the developer's head to understand what they need to accomplish; as such, the quality of their results is heavily dependent on the quality of the model they use.

An exciting direction for RSSEs is the possibility of proactive discovery. Rather than waiting for the developer to realize that they need a certain kind of information, it gets delivered to them beforehand. The challenge is to avoid overwhelming the developer with so many "helpful" hints that these are all ignored. Models that balance adaptation to the developer's actions, with reaction to the developer's feedback and to the developer's stated preferences would seem to hold out the most promise but also the greatest challenge. To date, the predominant output mode for RSSEs has been the simple recommendation list. Recommendation lists have many limitations, however, especially when it comes to explaining the results of the recommendation systems. Strathcona takes a departure from the standard model, with additional views to graphically represent the internal structure of the recommended examples (see Figure 1).

The evolution of RSSEs is not only influenced by the needs of developers, but also by the nature of available data and by the development of technologies. The *2008 International Workshop on Recommendation Systems for Software Engineering* allowed us to observe new trends and to hear numerous well-informed opinions about the future of the field. To date, the majority of RSSEs have

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

focused on recommendations related to software development artifacts, and source code in particular. RSSEs typically recommend code; e.g., code to look at, code to change, or code to reuse. However, recommendations can be provided about many other aspects of software development [4], including quality measures, tools, project management, and people, to support an ever-widening array of software engineering tasks. One of the early RSSEs, Expertise Browser [7], was designed to help developers find people with the expertise required to answer their question. Since then, however, few recommendation systems have specifically focused on recommending people. With the recent popularity of social networks in software development [11], the tide appears to be reversing, with a new generation of RSSEs (such as Dhruv [2]) specifically designed to recommend people that developers should interact with to succeed at their task.

As recommendation systems for software engineering continue to be developed and adopted, we are bound to see new ways for systems to support the work of developers, by cost-effectively recommending for them information that is novel, surprising, and relevant.

Acknowledgments

The authors are grateful to Barthélémy Dagenais, Rob DeLine, and Reid Holmes as well as the anonymous reviewers for their insightful comments on this paper.

SIDEBAR—How Do Recommendation Systems for Software Engineering Relate to Search Tools?

Search tools constitute a class of systems that feels similar to RSSEs. To appreciate the relationship, one must first notice that our definition of RSSEs is *inclusive*: an RSSE may provide other functionality beyond recommendations, and its recommendation features may not even be its most prominent ones. Thus, a particular search tool might also be an RSSE, but not all search tools are RSSEs, nor are all RSSEs also search tools.

Classic search tools like grep (a regular expression matcher) are not RSSEs. Grep contains no model of the context and software engineering task at hand. Grep always returns precisely what is requested: thus, the developer must drive it very carefully to avoid poor results.

To address this burden, more modern search engines have begun to include recommendation features. Standard web search engines order their results based on a general-purpose model of relevance. Source code search engines like Google Code Search, Krugle, or Koders specialize this idea for searching through source code repositories, but do not possess any model of the developer's task or context. As with grep, code search engines can be used for specific means to an end, but they do not eliminate the essential difficulty of translating a complex software engineering task into concrete search terms.

In the end, if you know exactly what you are looking for and you know exactly how to specify it, a search tool is likely to be the right fit. In other cases, the right recommendation system for software engineering ought to focus your efforts.

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

SIDEBAR—Some Other RSSEs

We give a brief look at a few other RSSEs. There are definitely many more around; for further information, see our RSSE community website at http://sites.google.com/site/rsseresearch/.

CodeBroker

CodeBroker [13] is a tool that analyzes comments in the developer's code in order to detect similarity to elements in a class library that could likely be used to implement the described functionality. CodeBroker uses a combination of textual similarity analysis and type signature matching to identify relevant elements. It works in push mode, producing recommendations every time a comment is written; it also manages user-specific lists of "known components", which are automatically removed from the recommendations.

Dhruv

Dhruv [2] recommends people and artifacts relevant to a bug report, chiefly aimed at the open-source community that interacts heavily via the Web. It uses a three-layer model of community (developers, users, contributors), content (code, bug reports, forum messages), and interactions between these. A Semantic Web is constructed that describes the objects and relationships between them; objects are then recommended according to the similarity between the bug report and the terms contained within the object plus its meta-data.

Expertise Browser

Finding the people with the right expertise is a difficult task in software development, especially when developers are geographically distributed. Expertise Browser [7] is a tool that recommends who has appropriate expertise for a given code location or documentation. Recommendations are based on past changes, developers who changed a method are assumed to have expertise for that method.

PARSEWeb

There are times when you want to call a particular method, but you have trouble figuring out how to access it with an appropriate calling sequence. PARSEWeb [12] recommends call chains on the basis of how often they occur in web-based repositories of example code. Example code is preprocessed to identify frequently occurring patterns of calls; patterns that result in the execution of the developer's target method are recommended.

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

References

[1] ACM International Conference on Recommender Systems. http://recsys.acm.org, 2009.

[2] A. Ankolekar, K. Sycara, J. Herbsleb, R. Kraut, and C. Welty. Supporting online problem-solving communities with the Semantic Web. In *Proceedings of the International Conference on the World Wide Web*, pages 575–584, 2006.

[3] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*, pages 481-490, 2008.

[4] H.-J. Happel and W. Maalej. Potentials and challenges of recommendation systems for software development. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, pages 11–15, 2008.

[5] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach for recommending relevant examples. *IEEE Transactions on Software Engineering*, 32(1):952–970, 2006.

[6] J. A. Konstan, J. Riedl, B. Smyth, F. J. Martin, and P. Pu. Foreword. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, page iii, 2007.

[7] A. Mockus and J. D. Herbsleb. Expertise Browser: A quantitative approach to identifying expertise. In *Proceedings of the International Conference on Software Engineering*, pages 503–512, 2002.

[8] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452-461, 2006.

[9] M. P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology*, 17(4), August 2008.

[10] M. P. Robillard, R. J. Walker, and T. Zimmermann. International Workshop on Recommendation Systems for Software Engineering. http://pages.cpsc.ucalgary.ca/˜zimmerth/rsse-2008/, 2008.

[11] M. Swaine. Social networks and software development. *Dr. Dobb's magazine*, February 2008. http://www.ddj.com/architect/206104412.

[12] S. Thummalapenta and T. Xie. PARSEWeb: A programming assistant for reusing open source code on the Web. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213, 2007.

[13] Y. Ye and G. Fischer. Reuse-conducive development environments. *Automated Software Engineering*, 12(2):199–235, 2005.

[14] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

Authors' bios

Martin P. Robillard is an associate professor in the School of Computer Science at McGill University in Montréal, Canada. His research focuses on software evolution and maintenance. He received his Ph.D. in Computer Science from the University of British Columbia in 2004. Contact him at martin@cs.mcgill.ca

Robert J. Walker is an associate professor in the Department of Computer Science at the University of Calgary in Calgary, Canada. His research interests focus on software evolution and reuse. He received his Ph.D. in Computer Science from the University of British Columbia in 2003. Contact him at walker@ucalgary.ca

Thomas Zimmermann is a researcher at Microsoft Research, where he works in the Empirical Software Engineering and Measurement area. His research focus is the evolution of large and complex software systems. He conducts empirical studies and builds tools that use data mining to support programmers. Zimmermann received his PhD in computer science from Saarland University in Germany. Contact him at tzimmer@microsoft.com

Contact Information

Martin Robillard
School of Computer Science
McGill University
#318-3480 University Street
Montréal, QC H3A 2A7
Canada

Tel.: (514) 398-4258
Fax.: (514) 398-3883
email: martin@cs.mcgill.ca

Robert J. Walker
Department of Computer Science
University of Calgary
2500 University Dr. NW
Calgary, AB T2N 1N4
Canada
Tel.: +1.403.210.9593
Fax: +1.403.284.4707
email: walker@ucalgary.ca

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Software but has not yet been fully edited.
Some content may change prior to final publication.

Thomas Zimmermann
Microsoft Research
One Microsoft Way
Redmond, WA 98052 USA
Email: tzimmer@microsoft.com
Office: (425) 703-8450
Fax: (425) 936-7329

Digital Object Indentifier 10.1109/MS.2009.161 0740-7459/$26.00 © 2009 IEEE
Authorized licensed use limited to: West Virginia University. Downloaded on October 30, 2009 at 20:28 from IEEE Xplore. Restrictions apply.