Recommendation Systems for Software Engineering

Martin P. Robillard, *McGill University*

Robert J. Walker, *University of Calgary*

Thomas Zimmermann, *Microsoft Research*

Recommendation systems specific to software engineering are emerging to assist developers in a wide range of activities. This overview of available systems describes what they are, what they can do now, and what they might do in the future.

Despite steady advancement in the state of the art, software engineering remains a challenging endeavor. Developers are continually introduced to new technologies, components, and ideas. The systems they 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 task as mundane as adding a message to a status bar might involve discovering the right classes among thousands in a class library and then understanding the complex interactions among them. Without assistance, it's easy to become bogged down in a morass of details and spend a disproportionate amount of time seeking information at the expense of other value-producing tasks.

Various recommendation systems help people find information and make decisions where they lack experience or can't consider all the data at hand. These systems combine many computer science and engineering methods to proactively tailor suggestions that meet users' particular information needs and preferences.[1] To date, most recommendation systems have been tied to the Web. Many of them embody mature technology delivered as part of commercial systems, such as Amazon.com's recommenders.

The challenges people face in navigating large information spaces have similarities to those of software developers trying to find the one class they need from a library of hundreds. Recommendation systems for software engineering (RSSEs) are emerging to assist developers in various activities—from reusing code to writing effective bug reports. In addition to the scale of software systems and libraries, the increasing pace of their evolution and extent of their heterogeneity are also driving RSSE development. Likewise, the increase in distributed development constrains knowledge sharing among team members, motivating technological alternatives.

Key factors giving rise to practical RSSEs include large stores of publicly available source code for analyzing recommendations, mature software-repository mining techniques, and mainstream adoption of common software development interfaces, including Web interfaces such as Bugzilla and tool-integration platforms such as Eclipse.

RSSEs are ready to become part of industrial software developers' toolboxes. Research prototypes are quickly maturing, tools are being released, and first-generation systems are being reimplemented in different environments.[2] In this overview, we describe what RSSEs are, what they can do for developers, and what they might do in the near future.

RSSEs: What They Are

We start with a general definition and description proposed by the organizers of the ACM International Conference on Recommender Systems (RecSys 09; http://recsys.acm.org/2009):

> *[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.*

RSSEs match this definition in their aim to support developers in decision making. Particularly with respect to information-seeking goals, RSSEs help developers find the right code, API, and human expert in information spaces comprising a system's code base, libraries, bug reports, version history, and other documentation.

RSSEs also relate their output to a user's interests. Developers can express their interests explicitly through a direct query or implicitly through actions that the RSSE factors into its recommendations. This distinction highlights a general challenge for recommendation systems—namely, how to establish context, which could include all relevant information about the user, his or her working environment, and the project or task status at the time of the recommendation.

In recommendation systems for traditional domains, the context is typically established through a user profile, which can consist of any combination of user-specified and learned characteristics. However, the context for RSSEs includes the comparatively rich range of activities associated with software development tasks. A traveler who seeks a hotel recommendation can typically specify much of the context with a handful of simple criteria, such as price, amenities, star-rating, distance from area of interest. In contrast, a software developer who wants help finding where to look next when exploring source code needs a recommendation system that can establish several rather fuzzy parameters, such as what the developer already knows and which parts of the source code are related to his or her needs.

An RSSE might need to provide or infer all the following aspects as part of the context:

- *the user's characteristics*, such as job description, expertise level, prior work, and social network;
- *the kind of task being conducted*, such as adding new features, debugging, or optimizing;
- *the task's specific characteristics*, such as edited code, viewed code, or code dependencies; and
- *the user's past actions or those of the user's peers*, such as artifacts viewed and artifacts explicitly recommended.

The last part of the RecSys 09 general description addresses qualities of system output: novelty, surprise, and relevance. To assist software developers, RSSEs must provide information that's relevant to their problem and useful to them. The recommendations must be both situation- and user-specific. Sometimes a recommendation is valuable because the developer wasn't aware of a need or all the risks it posed. Sometimes it's valuable merely because it corroborates the developer's suspicions—for example, confirming a thought that a call to `Document.getContentSize()` is the only dependency on JDOM (Java document object model) within a project.

Considering all these particulars of software engineering, we've developed the following definition:

> *An RSSE is a software application that provides information items estimated to be valuable for a software engineering task in a given context.*

How to establish context is a general challenge for recommendation systems.

What RSSEs Do for Developers

By helping developers find information they should know about and evaluate alternative decisions, RSSEs span a wide spectrum of software engineering tasks and practically unbounded amounts of development data.

Most current RSSEs support developers while programming. For example, CodeBroker[3] has demonstrated its potential for surfacing reuse opportunities, and Expertise Browser[4] has done the same for locating expert consultants. Available RSSEs also facilitate deciding what examples to use (Strathcona[5]) and what call sequences to make (ParseWeb[6]). They can help navigate large code bases by boiling down rich and complex information spaces into clearly prioritized lists

Preprocessing the project's CVS archive identifies changes to program elements such as classes.

of alternatives, such as where to look in the code (Suade[7]) and what to change next (eRose[8]).

Beyond programming support, current RSSEs can recommend replacement methods for adapting code to a new library version (SemDiff[9]). They can also help during debugging by finding code and people related to a bug fix (Dhruv[10]). Additionally, they can predict which parts of a software product will have the most defects and thus help prioritize test resources for quality assurance.[11]

This diversity makes generalizations about RSSE architectures difficult, but most involve at least three main functionalities:

- *a data-collection mechanism* to collect development-process data and artifacts in a data model;
- *a recommendation engine* to analyze the data model and generate recommendations; and
- *a user interface* to trigger the recommendation cycle and present its results.

To illustrate what RSSEs can do, we present three example systems that we've developed and experimented with. The examples don't represent the entire field, but they do show important differences between RSSEs.

Guiding Software Changes with eRose

If you browse books at Amazon.com, you can encounter recommendations of the form, "Customers who bought this book also bought…." Such suggestions stem from purchase history. Buying two or more books together establishes a relationship between them, which Amazon.com uses to create recommendations.

The eRose plug-in[8] for the Eclipse integrated development environment (IDE) realizes a similar feature for software development by mining past changes from version archives, such as Concurrent Versions System (CVS). This feature tracks changed elements (the context) and updates recommendations in a view after every save operation. For example, if a developer wants to add a new preference to the Eclipse IDE and so changes `fKeys[]` and `initDefaults()`, eRose would recommend "Change `plugin.properties`" because all developers who changed the Eclipse code 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, it groups elements that were changed at the same time and by the same developer ("co-changed") into transactions and stores them in an SQL database. At runtime, eRose uses the developer's context to query the database for transactions that contain at least one of the context elements. It then extracts the elements from these transactions to derive recommendations.

In deriving recommendations from these results, eRose first excludes elements in the context, because the user has already changed them. 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 eRose's underlying concept is co-change, it's fairly language independent and can recommend text, image, or documentation files in addition to program elements. Furthermore, eRose can reveal hidden dependencies. For example, when a developer changes code to create a database, he or she might also need to update the diagram file depicting the database schema.

A prototype implementation of eRose is available at www.st.cs.uni-saarland.de/softevo/erose.

Finding Relevant Examples with Strathcona

Frameworks give developers a code repository that can help in their coding tasks. However, frameworks are frequently large and difficult to understand, and documentation is often incomplete or otherwise insufficient to help with specific tasks.

The Strathcona system[5] retrieves relevant source code examples to help developers use frameworks effectively. For example, a developer who's trying to figure out how to change the status bar in the Eclipse IDE can highlight the partially complete code (the context) and ask Strathcona for similar examples, as shown in Figure 1a. Strathcona extracts a set of structural facts from the code fragment, such as what types are referenced (`IStatusLineManager`, an abstract interface type) and what methods are called (`setMessage(String)`).

Strathcona uses PostgreSQL queries to search for occurrences of each fact in a code repository. Next, it uses a set of heuristics to decide on the best examples, which it orders according to how many heuristics select them. It returns the top 10 examples, displaying them in two formats—a structural overview (Figure 1b) diagram and highlighted source code (Figure 1c)—to show similarities to the developer's partially complete code. Developers can also view a rationale for a proposed example, as shown in Figure 1d.

A prototype and more details are available at http://lsmr.cs.ucalgary.ca/strathcona.

Guiding Software Navigation with Suade

Suade is an Eclipse plug-in that automatically

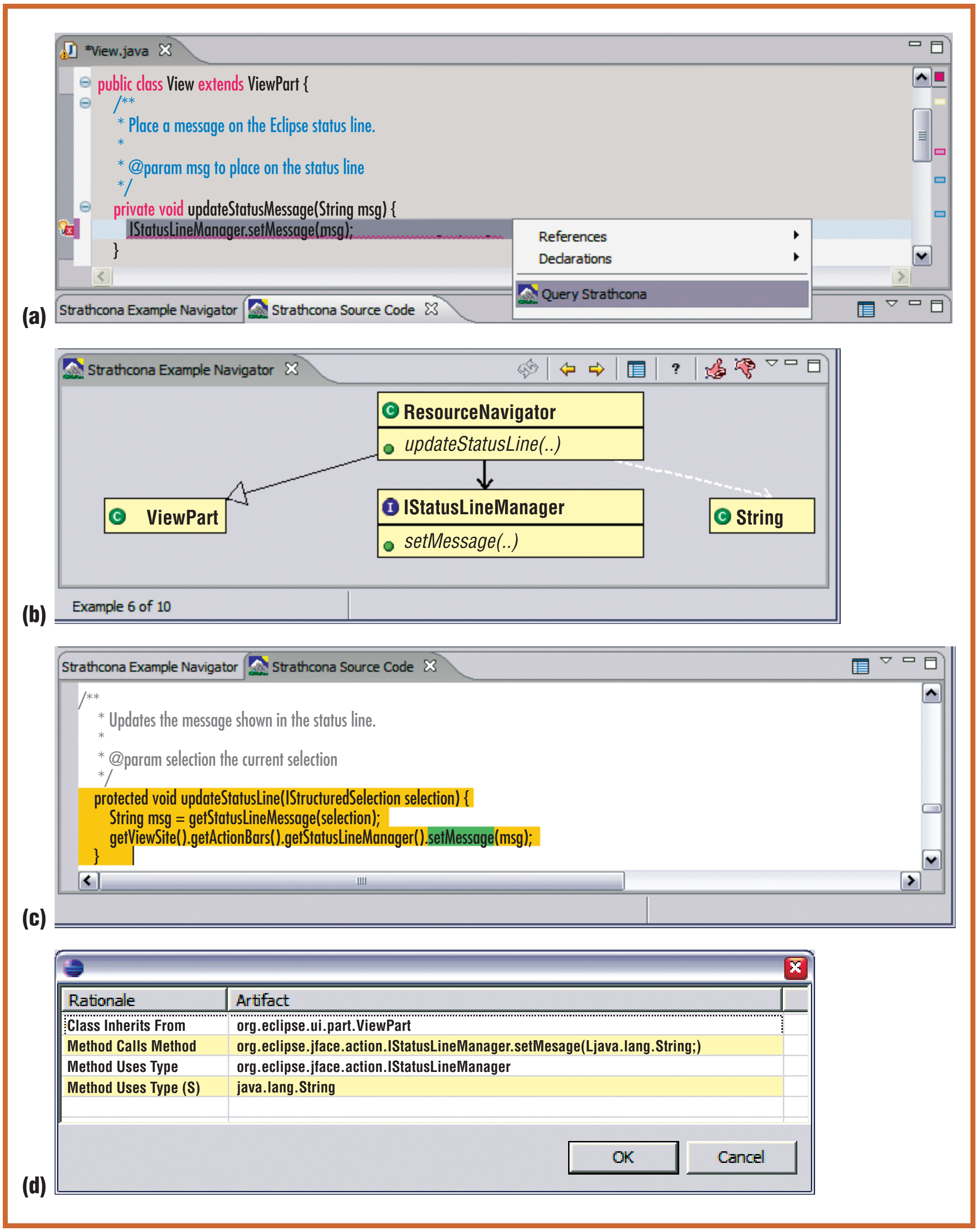


Figure 1. Strathcona user interface: (a) recommendation query; results in (b) a structural overview and (c) highlighted source code; and (d) rationale for recommendation. The multiple output views let developers quickly assess the potential relevance of each example before wading into the details.

generates suggestions for software investigation.[11] Developers who become stuck while exploring code to complete a change task can use Suade to trigger recommendations about where to look next among all the related elements.

The developer explicitly specifies a set of relevant fields and methods (the context elements), and Suade uses method-call and field-access relations to automatically retrieve related elements. It ranks the retrieved elements by extracting a dependency graph of all their static dependencies from the project's source code to the context elements, and then by applying heuristics to the graph's topology. For example, if a method calls only those methods that a developer specified as relevant, it's ranked higher than methods that call the context methods in addition to many others.

In Suade, users create a context by dragging and dropping elements of interest into a view. Once they've specified a context, they can trigger a recommendation cycle. Suade displays recommendations as a list in a dedicated view. Users can drag recommended elements back into the context view to iteratively update recommendations.

Some Other RSSEs

The main text refers explicitly to four recommendation systems for software engineering (RSSEs) that we summarize here, but there are many more. For further information, see our RSSE community website at http://rsse.org.

CodeBroker

CodeBroker[1] analyzes developer comments in the code to detect similarities to class library elements that could help 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 developer writes a comment. It also manages user-specific lists of "known components," which it automatically removes from its recommendations.

Dhruv

Dhruv[2] recommends people and artifacts relevant to a bug report. It operates chiefly in the open source community, which interacts heavily via the Web. Using a three-layer model of community (developers, users, and contributors), content (code, bug reports, and forum messages), and interactions between these, Dhruv constructs a Semantic Web that describes the objects and their relationships. It recommends objects according to the similarity between a bug report and the terms contained in the object and its metadata.

Expertise Browser

Finding the right software experts to consult can be difficult, especially when they're geographically distributed. Expertise Browser[3] is a tool that recommends people by detecting past changes to a given code location or document. It assumes that developers who changed a method have expertise in it.

ParseWeb

Sometimes you might want to call methods on an object of a particular type but you don't know how to obtain objects of that type from objects available in your programming context (for example, method parameters). ParseWeb[4] recommends sequences of method calls starting from an available object type and producing a desired object type. ParseWeb analyzes example code found on the Web to identify frequently occurring call patterns that link available object types with desired object types. Developers use the tool by specifying available and desired object types and requesting recommendations.

References

1. Y. Ye and G. Fischer, "Reuse-Conducive Development Environments," *Automated Software Eng.*, vol. 12, no. 2, 2005, pp. 199–235.
2. A. Ankolekar et al., "Supporting Online Problem-Solving Communities with the Semantic Web," *Proc. Int'l Conf. World Wide Web*, ACM Press, 2006, pp. 575–584.
3. A. Mockus and J.D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," *Proc. Int'l Conf. Software Eng.* (ICSE 02), IEEE CS Press, 2002, pp. 503–512.
4. S. Thummalapenta and T. Xie, "PARSEWeb: A Programming Assistant for Reusing Open Source Code on the Web," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 07), ACM Press, 2007, pp. 204–213.

More information on Suade, including screenshots and a downloadable version of the tool, is available at www.cs.mcgill.ca/~swevo/suade.

RSSE Design Dimensions

The examples we've described represent only a small fraction of the RSSEs available and in development. To take a broader look at the field, we now consider three RSSE design dimensions: nature of the context, recommendation engine, and output modes. Table 1 summarizes these dimensions, and the sidebar describes some other RSSEs we mention in the discussion.

Nature of the Context

The recommendation context is a core RSSE concept. It's the RSSE input, and it can be explicit, implicit, or a hybrid of these strategies.

Developers can provide context explicitly through various user-interface interactions, such as 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 explicitly is appropriate for contexts that are difficult to detect, such as the user's interest or experience level, and where the burden of specification is small. For many tasks, RSSEs can obtain part of the context implicitly. For example, some systems can track and react to developer actions (as in eRose), and some require context information that would be unreasonable to specify explicitly (such as a developer's interaction history with the IDE).

Finally, many cases will require a combination of implicit and explicit context gathering. With Strathcona, the developer explicitly selects a section of code text, but the system parses and analyzes the code to implicitly extract a structured model of the RSSE's context. CodeBroker automatically extracts the context from code comments, a syntactic construct that it assumes to be part of the task.

Recommendation Engine

RSSEs must analyze more than context data to make their recommendations. Additional data can include the project's source code, the complete history of system changes, artifacts such as emails posted to mailing lists and bug reports, interaction data accumulated from many programming sessions, test coverage reports, and code bases external to the project. Analysis of these data sources, often referred to as mining software repositories (MSR), was a theme topic of a recent *IEEE Software* special issue (Jan./Feb. 2009). MSR is just one potential means to an end, and not all RSSEs rely on it to produce recommendations (for example, Suade does not).

Every RSSE we've encountered uses a ranking mechanism as a cornerstone of its analysis. An ideal ranking algorithm systematically puts the items most valuable to the user at the top of its rankings. In practice, rankings rely on a model of what a developer will find useful. 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 one developer might not be useful for his or her colleague.

Models used by recommendation engines might also have to account for time sensitivity: what's useful for a developer now might not have been useful in the past or might not be useful in the future.

Table 1

RSSE design dimensions

Nature of the context	Recommendation engine	Output mode
Input: explicit | implicit | hybrid	**Data:** source | change | bug reports | mailing lists | interaction history | peers' actions	**Mode:** push | pull
	Ranking: yes | no	**Presentation:** batch | inline
	Explanations: from none to detailed	
User feedback: none | locally adjustable | individually adaptive | globally adaptive		

Output Modes

Most existing RSSEs operate in pull mode and produce recommendations after a developer's explicit requests, which can be as simple as a single click in an IDE. Some RSSEs operate in push mode, delivering results continuously (for example, eRose, CodeBroker, and Dhruv). Push mode can be obstructive if it isn't designed well. Conversely, developers can miss something important in pull mode if they don't even think to ask about it.

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

Cross-Dimensional Features

RSSE features can cross design dimensions. For example, the recommendation engine can take the developer's interactions with the RSSE into account, allowing the developer to flag bad recommendations to eliminate them from future results. In this way, past recommendations support a feedback mechanism and become part of the context or data on which the RSSE operates.

Ranking mechanisms can be locally adjustable (the developer adjusts the inferred context manually, as in Suade); individually adaptive (the algorithm is refined for individuals according to their implicit or explicit feedback, as in CodeBroker); or globally adaptive (feedback from one user affects another user). Existing RSSEs are often limited in the ranking mechanisms they offer.

Finally, RSSEs vary in how they explain their results. At one extreme, some recommendations appear to be almost magical—for example, predicting that certain files will be defect-prone with no explanation. Developers will have a hard time trusting these recommendations. At the other extreme are systems such as Strathcona that provide detailed rationales justifying each recommendation. Naturally, such detailed rationales expose part of the RSSE's inner workings, which has both potential benefits, such as increasing confidence in the recommendation, and pitfalls, such as information overload.

RSSE Limitations and Potential

RSSEs advance the state of the art in software development tools, but they aren't without their limitations. For example, when information repositories are large, RSSEs can face the "cold-start problem" of lacking enough information to make recommendations until the project is underway. One solution to this problem is to leverage analogous data from other projects. Additionally, because RSSEs can't crawl inside developers' heads to understand what they need to accomplish, the results' quality depends heavily on the quality of the RSSE's model.

Proactive discovery is an exciting direction for future RSSEs. Rather than waiting for developers to realize they need a certain kind of information, the system would deliver it to them automatically. The challenge is to avoid giving so many "helpful" hints that the developer finally ignores them all.

Models that balance adaptation to developers' actions with reaction to their feedback and stated preferences seem the most promising but also the most challenging. To date, the predominant RSSE output mode has been a simple recommendation list. Such lists have many limitations, however, especially when it comes to explaining the results. Strathcona departs from the standard model by offering graphical representations of the recommended examples' inner structures (see Figure 1).

RSSEs evolve not only with developers' needs but also with the nature of available data and development of technologies.[2] To date, most RSSEs have focused on recommendations related to software development artifacts, particularly source code. RSSEs typically recommend code—to look at, change, or reuse. However, recommendations could address many other aspects of software development.[12] For example, recommendations for quality measures, tools, project management, and people could support an ever-widening array of software engineering tasks. One of the early RSSEs, Expertise Browser,[4] was designed to help developers find people with the expertise to answer their questions. Since then, however, few recommendation systems have adopted this focus. With the recent popularity of social networks in software development,[13] the tide appears to be reversing and a new generation of RSSEs, such as Dhruv,[10] can recommend people that developers should interact with to succeed at their task.

As RSSEs continue to develop, we're bound to see new ways for systems to support developers by cost-effectively recommending information that's novel, surprising, and relevant.

Acknowledgments

We're grateful to Barthélémy Dagenais, Rob DeLine, and Reid Holmes as well as the anonymous reviewers for their insightful comments on this article.

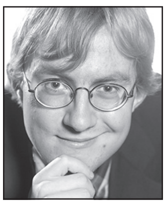
References

1. J.A. Konstan et al., "Foreword," *Proc. 2007 ACM Conf. Recommender Systems* (RecSys 07), ACM Press, 2007. p. iii.
2. M.P. Robillard, R.J. Walker, and T. Zimmermann, "Foreword," *Proc. Int'l Workshop on Recommendation Systems for Software Engineering*, ACM Press, 2008; www.rsse.org.
3. Y. Ye and G. Fischer, "Reuse-Conducive Development Environments," *Automated Software Eng.*, vol. 12, no. 2, 2005, pp. 199–235.
4. A. Mockus and J.D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," *Proc. Int'l Conf. Software Eng.* (ICSE 02), IEEE CS Press, 2002, pp. 503–512.
5. R. Holmes, R.J. Walker, and G.C. Murphy, "Approximate Structural Context Matching: An Approach for Recommending Relevant Examples," *IEEE Trans. Software Eng.*, vol. 32, no. 1, 2006, pp. 952–970.
6. S. Thummalapenta and T. Xie, "PARSEWeb: A Programming Assistant for Reusing Open Source Code on the Web," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 07), ACM Press, 2007, pp. 204–213.
7. M.P. Robillard, "Topology Analysis of Software Dependencies," *ACM Trans. Software Eng. and Methodology*, vol. 17, no. 4, 2008, article no. 18.
8. T. Zimmermann et al., "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, 2005, pp. 429–445.
9. B. Dagenais and M.P. Robillard, "Recommending Adaptive Changes for Framework Evolution," *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), IEEE CS Press, 2008, pp. 481–490.
10. A. Ankolekar et al., "Supporting Online Problem-Solving Communities with the Semantic Web," *Proc. Int'l Conf. World Wide Web*, ACM Press, 2006, pp. 575–584.
11. N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," *Proc. 28th Int'l Conf. Software Eng.* (ICSE 06), IEEE CS Press, 2006, pp. 452-461.
12. H.-J. Happel and W. Maalej, "Potentials and Challenges of Recommendation Systems for Software Development," *Proc. Int'l Workshop on Recommendation Systems for Software Eng.* (RSSE 08), ACM, 2008, pp. 11–15.
13. M. Swaine, "Social Networks and Software Development," *Dr. Dobb's*, Feb. 2008; www.ddj.com/architect/206104412.

About the Authors

Martin P. Robillard is an associate professor in McGill University's School of Computer Science. His research focuses on software evolution and maintenance. Robillard received his PhD in computer science from the University of British Columbia. Contact him at martin@cs.mcgill.ca.

Robert J. Walker is an associate professor in the University of Calgary's Department of Computer Science. His research interests focus on software evolution and reuse. Walker received his PhD in computer science from the University of British Columbia. Contact him at walker@ucalgary.ca.



Thomas Zimmermann is a researcher in empirical software engineering and measurement at Microsoft Research. His research focuses on the evolution of large, complex software systems and involves conducting empirical studies and building 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.