Recent Advances in Recommendation Systems for Software Engineering

Robert J. Walker

University of Calgary
Calgary, Canada
walker@ucalgary.ca

Abstract. Software engineers must contend with situations in which they are exposed to an excess of information, cannot readily express the kinds of information they need, or must make decisions where computation of the unequivocally correct answer is infeasible. Recommendation systems have the potential to assist in such cases. This paper overviews some recent developments in recommendation systems for software engineering, and points out their similarities to and differences from more typical, commercial applications of recommendation systems. The paper focuses in particular on the problem of software reuse, and speculates why the recently cancelled Google Code Search project was doomed to failure as a general purpose tool.

Keywords: RSSEs, overview, classification, opportunities.

1 Introduction

At one time, the notion of a recommendation system for software engineering (RSSE) would have been anathema: computers and software were all about correctness and precision, while recommendations bespeak fuzziness and faith. Unfortunately, many tasks that face the software engineer are inherently imprecise, staggeringly complex, and often infeasible to compute (or even formally undecidable). The software engineer must bridge the gaps between: the messy, changing needs of the real world; the interpersonal and organizational challenges involved in collaborating with other people; and the inflexibility of the computer. While correctness and precision have their place, a willingness to revert to "good enough" answers can be the difference between success and failure in the real world with its constraints on time and money.

Robillard et al. [14] have previously defined RSSEs as follows: "*An RSSE is a software application that provides information items estimated to be valuable for a software engineering task in a given context.*" A key word in that definition is "estimated", because if the correct answer can be faithfully returned in all circumstances, the application is not making recommendations but computing or finding *the* answer. For example, a search for all references to a particular method within a software project is not a recommendation: there is a unique, unequivocally correct set. In contrast, finding all locations in a system that should be changed when that method is changed will require recommendations, due to the infeasibility of computing the correct answer in general.

Recommendations for a software engineer play much the same role as recommendations for any user. There is a problem or task at hand. There is (often highly structured) information available about potential solutions. Some additional information about the specific task is often available—e.g., through an integrated development environment (IDE)—as is information about what other people have done in similar circumstances. The recommendation system's task is to infer the nature or details of the task, the needs and/or characteristics of the context of the task and of the engineer, to construct one or more recommendations according to some model of relevance, and to present these recommendations in a manner conducive to their timely and efficient usage.

An analysis of the differences and similarities of different recommendation systems can be facilitated by a classification scheme. Section 2 outlines three different schemes for classifying recommenders in general recommenders or RSSEs in particular: the paradigmatic classification for general recommenders provided by Jannach et al. [9], the multidimensional design classification that provided by Robillard et al. [14] for RSSEs, and a novel problem-space classification.

Section 3 examines a small set of recent RSSEs, to illustrate the breadth of applications that have been explored. Section 4 focuses more narrowly on software reuse, being a cornerstone of the author's research and a fundamental goal that led to the field of software engineering. Section 5 points to some opportunities with RSSEs that the author has seen in his own work. Section 6 summarizes and points out remaining challenges and opportunities in RSSE and potential cross-fertilization with research in recommenders from other domains.

2 Categorizing RSSEs

Three categorization schemes are presented here: the paradigmatic classification for general recommenders provided by Jannach et al. [9], the design-oriented classification for RSSEs developed by Robillard et al. [14], and a novel problem-space classification.

Jannach et al. [9] divide recommenders into four paradigms. Their stated focus is personalized recommendations (i.e., ones reflective of the individual's tastes, needs, habits, or characteristics) rather than generic recommendations. The archetypal example is of the online bookstore, where recommendations are made to a customer about what books may be of interest to them.

1. *Collaborative recommenders.* A typical example is, if Alice has purchased or viewed books that have been purchased or viewed by other customers, additional books purchased or viewed by those other customers are also of potential interest to Alice. Collaborative recommenders can thus be viewed as operating through analogical reasoning over different users, and extrapolation from historical data, without resorting to detailed analysis of the items being recommended.

2. *Content-based recommenders.* Again, considering the online bookstore archetype, recommendations could be derived from the properties of the books that Alice has purchased or viewed: the author, the genre, the publication date, keywords in the title or synopsis, . . . all are potential signs of what Alice cares about. Content-based recommenders then realize a similarity function (which is also a form of analogical reasoning) that depends on a specific model of the items being recommended and extrapolation from historical data.
3. *Knowledge-based recommenders.* When Alice can express properties of the books she is interested in or some of her individual properties (e.g., age, gender), the need for historical data about Alice might be avoidable. Knowledge-based recommenders also realize a similarity function that depends on a specific model of the items being recommended. The key difference from content-based recommenders is the avoidance of historical data.
4. *Hybrid recommenders.* Since each of the above approaches has advantages and disadvantages, a good combination of them would draw on their strengths while cancelling their weaknesses. Of course, a bad combination would do the opposite.

Robillard et al. provide a table of the design dimensions for RSSEs, reproduced in Table 1. The major dimensions focus on three issues. The *context* consists of the situation-specific information to be used by the recommender, in the form of implicit input garnered by observing the developer, explicit input provided by the developer, or some hybrid of the two. To construct its recommendations, the recommendation engine analyzes the context plus additional sources of *data*, such as the source code of a project being operated on (beyond an explicit input), a repository of change history information, or websites containing developer conversations. While the majority of RSSEs rank their results, not all do so. The *output* involves two aspects: the mode of the recommendations may be via push, where the developer has not actually asked for help, or via pull, where an explicit request was made; the presentation style may insert the recommendations within another source of information, like the developer's standard views in an IDE, or all at once in a special-purpose display. Rationales (or explanations) for the recommendations vary in nature and detail a great deal, and involve an interplay between the recommendation engine itself and the output. And finally, all manner of developer feedback can potential be accepted to refine or to change the recommendations, involving an individual's feedback affecting just their local view or everyone's.

A software engineer encounters three kinds of situations in which recommendations could potentially be beneficial.[1] (1) In the presence of too much information, recommendations of the most relevant cases can save time and prevent those cases from being overlooked—*the information overload situation.* (2) In the absence of specific knowledge about what information to seek or how to express it, recommendations can draw on the current context (of the engineer) and

[1] Note that this categorization is not derived from a systematic analysis of RSSEs, so there may be other cases that do not fit here, but these three cases do occur.

Table 1. Design dimensions of RSSEs according to Robillard et al. [14]

Context nature	**Recommendation engine**	**Output form**
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		

the past experiences of others—*the navigation confusion situation.* (3) When the correct answer to a problem is too complex to compute, a set of heuristically-based answers or approximations can still help the engineer to make a rational decision—*the infeasible computation situation.*

3 Sampling Recent RSSE Approaches

RSSEs have been realized for most activities within software engineering, from assisting with implementations, to pointing out high-risk code, to recommending people as experts, to helping plan out staged releases of features. Three specific examples are examined here.

In requirements elicitation, the problem is to discover each stakeholder's needs, preferences, and wishes in a form that can be compared and contrasted. A negotiation process then proceeds in order to resolve conflicts and ambiguities, and to reach agreement on rationales and priorities. Performing such activities in a distributed development environment is particularly challenging; in large systems, much time can be wasted involving individuals in negotiations on issues not relevant to them. The approach of Castro-Herrera et al. [2] seeks to analyze natural language descriptions of stakeholder's perspectives, in order to help clarify those perspectives and to cluster the stakeholders into specific themes for discussion. They utilize data describing the stakeholders for this purpose as well, and user feedback is used to improve the clustering process. This is a hybrid approach, involving elements of collaborative recommenders and knowledge-based recommenders. It can be seen as coping with an information overload problem (the need to read and understand the ideas of a large set of stakeholders) and a navigation confusion problem (the need to determine which stakeholders must focus on individual issues). The input context is explicit; the data source is stakeholder profiles; the output mode is pull; the presentation is batch; no explanations are provided; and the user feedback is individually adaptive, altering each user's own profile.

The YooHoo tool [6] tackles the problem of information overload for a developer whose development depends on external projects undergoing parallel development. At issue is the unpleasant surprise when one's software breaks due to an unexpected change event in an external project. While it is possible to monitor a stream of change events arriving from the external project, these will frequently be voluminous and mostly irrelevant. YooHoo attempts to classify the changes into those most likely to be impactful on the individual developer's project, those unlikely to be impactful, and those that are almost certainly irrelevant. To create its recommendations, YooHoo is configured to receive the relevant change event streams and analyzes the developer's project for external dependencies. The developer can also configure YooHoo to be more or less restrictive for particular event streams. YooHoo obtains over 99% elimination of events while maintaining a very high true positive rate. This approach deals strictly with an information overload problem. It is a knowledge-based approach. Its context is the preference settings of the developer; its other data sources are the developer's project and the external change event streams; its output is push mode (or perhaps calling it a "delayed-pull" mode would be better, since the developer has really asked for it) and batch; it does not use feedback.

Murphy-Hill et al. [12] have proposed a recommender for the unusual problem of command discovery within a developer's IDE. The issue is that modern IDEs like Eclipse suffer from a feature bloat problem: too many commands available for too much functionality. While feature-richness is important in an IDE, it can lead to inefficiencies when developers are unaware of the existence of a useful command, and they resort to more cumbersome combinations to achieve the same effect; since they are unaware of the existence of the command, there is no question of them searching for it. The approach operates atop a large repository of historical information about developers' command usage; commonality between the developer's actions and those of other developers prior to them learning about a command leads to the provision of a recommendation to use that command. The approach deals strictly with a navigation confusion problem. It is a collaborative recommendation approach. Its context is implicit in the actions of the developer; its other data source is the historical repository of command invocation sequences; its output is push mode and in-line; it does not use feedback.

4 RSSEs in Software Reuse

There have been many approaches over the years aimed at finding code for the developer. Traditional approaches focused on precise query languages to specify the desired semantics or other specific properties. The ultimate failure of these approaches hinges on their impracticality: industrial developers will rarely if ever be able or willing to spend the time crafting and debugging detailed, arcane specifications. The time required to do so is liable to outweigh the benefits accrued from finding the functionality. Furthermore, such approaches are likely to succeed in delivering relevant artifacts only in the simplest of cases; in others, no perfectly matching artifacts are likely to exist. Recognizing these problems, several researchers have followed the RSSE route to make progress.

4.1 Test-Driven Reuse

The test-driven development paradigm [1] argues that automated test cases should be developed before the code-under-test is implemented. The supposed benefits are: (1) automated test suites should be created anyways; (2) the developer will have a better understanding up-front of what that code ought to do; and (3) since it is natural to build assumptions (sometimes false) into our code, the developer will be less prone to also build those assumptions into their tests.

The idea of *test-driven reuse* is then to leverage these test cases in order to find pre-existing artifacts that can then be reused, instead of implementing the functionality from scratch. The core issue here falls into that of the infeasible computation problem: an unequivocally correct answer would require arbitrary transformations to arbitrary code that results in compilable versions of the examples that could then be run against the automated test cases; this is clearly infeasible in the general case, so providing approximations and recommendations is a reasonable alternative. Three research groups have pursued this idea [8, 10, 13]. Many of their ideas are essentially the same: extract a set of facts from the test cases, use these to find similar artifacts in a database, and recommend the most relevant artifacts to the developer—these are knowledge-based recommendation systems. The approach of Reiss [13] is slightly different, in that he also attempts some simple transformations of the located artifacts in order to be able to run the test cases on the (transformed) artifacts.

In unpublished research, all three techniques have been found to fail badly in most realistic circumstances, as they expect the developer's tests to use *exactly* the names of types and methods that are found in the artifacts. Even Reiss's approach falls into this trap, as he uses a strictly pipelined approach to select potential examples, only then transforming and running them, and it is the first step that contains the most significant limitation. As this investigation aimed to assess the relative strengths and weaknesses of the approaches, this resoundingly negative outcome related to strong work by serious researchers is surprising![2]

All three groups apparently viewed the task as a simple information retrieval problem: extract words $\rightarrow$ search for words, where the only real question was what model of similarity to use. There is little reason in general to expect that the developer will use the same words to describe their intentions as were used in the existing artifacts (the exception being in commonly recognized domains, and even there, there will be some propensity for variation) since they would not have already settled on any specific, existing frameworks.

[2] In fairness, Lemos et al. [10] emphasize that their approach aims specifically at locating "auxiliary functionality", but this distinction is not particularly clear.

Interestingly, commercial approaches to code search see much of the same problem: the desire to ignore the syntax and semantics of the source code, to treat it instead as just another kind of text. To be clear, this suffices for basic searches like "find me examples of the use of `IStatusLineManager`" but not "find me examples where there is a stack that ignores requests to insert the same element more than once." Google Code Search never overcame such limitations and never made it out of its beta release before the project was killed, despite years of incubation. This weakness could only have contributed to its downfall.

More generally, this issue points to important lessons for RSSEs. The research failed to recognize the needs of the developer to whom the recommendations are being given, and the nature of their task. It is important to remember the original, concrete point and to evaluate that, rather than assume that an abstraction maintains the significant properties.

4.2 Other Approaches for Finding Code

Other kinds of input have been used to help locate relevant artifacts for reuse. The specific problems being targeted can have significant impact on the suitability of a potential approach.

Strathcona. In some now-not-so-recent work, Holmes et al. [7] focused on the problem of finding examples of the use of specific abstract programming interfaces (APIs) such as are provided by libraries and frameworks. The issue at hand is one of information overload: real APIs are frequently large and complex, and to utilize their functionality can sometimes involve intricate protocols that are not always well documented. As a result, the developer can easily become lost as they attempt to understand how to solve a task through the application of an API. Examples are a common means for helping to overcome such issues, but handcrafted examples are expensive to create and require someone to have recognized a specific need.

Instead, the *skeleton* expressed by the developer (a partial and uncompilable implementation), and a large repository of existing implementations, can be leveraged. Note that, unlike in the case of test-driven reuse, it *is* reasonable here to assume that the types and methods being expressed in the skeleton are identical to those in the artifacts being leveraged: the developer is aware of which API they are using. A set of structural facts (e.g., which types are referenced, which methods are called) are extracted from the skeleton. A set of heuristics-based agents each seeks examples that are similar to the extracted facts, from the perspective of their individual approach. Examples that are ranked highly by many agents are ranked highest overall, and presented first to the developer.

Strathcona is a knowledge-based recommendation system, as it requires explicit input from the developer about the properties of the items of interest and leverages those properties to find potential examples. It does utilize an element of historical data: it can only provide examples that someone else has implicitly created in the past. It is not a collaborative recommendation system in the sense of Jannach et al. since there is no comparison between the developers themselves.

Strathcona addressed both the information overload problem and the navigation confusion problem. A key property of Strathcona that was unusual at the time was its provision of rationales for its recommendations: an explanation of which properties of the skeleton were met by a given example.

Mining feature descriptions. The recent approach of McMillan et al. [11] aims at supporting the rapid creation of prototypes: low-quality, partially-functional software useful in the assessment of the requirements and feasibility of a full-scale development effort. They argue that rapid prototyping consists of identification of candidate features (horizontal prototyping) followed by implementation of a subset of the feature set (vertical prototyping). Since the problem is really to invest as little time and money as possible to derive a sufficient prototype for stakeholders to tinker with, reusing existing software ought to reduce the cost of the process while arriving at an acceptable artifact.

Their idea is then to leverage the cursory descriptions typically created during horizontal prototyping to provide: (1) recommendations on the relevant features detected in the description that have also been detected in a repository of software components; and (2) recommendations on components providing those features that are deemed to not be too difficult to reuse (because they contain few external dependencies). Feedback to the feature recommendations are used to update the component recommendations.

As with Strathcona, this approach makes use of an existing, developer-oriented means of describing the entities of interest, rather than demanding that the developer conform to the needs of the software. Overall, this approach is a hybrid recommendation system: the initial input involves a knowledge-based recommender, but the resulting rating is a collaborative filtering approach. Using this to refine the identified features in the repository would also lead to improved results for other developers. As with Strathcona, the approach addresses both the information overload problem (finding components in a big repository) and the navigation confusion problem (who needs to talk to whom and about what).

5 Some Opportunities for RSSEs

The work in the Walker's lab has not always focused on RSSEs. The desire to compute the correct answer has often led to overlooking opportunities for more completely involving the developer. Two cases are presented here.

Jigsaw. If one wishes to reuse a software artifact within a partially existing system, the two will need to be integrated. The Jigsaw tool [5] largely automates this process for individual methods by inferring correspondences between the reused artifact's method and the target system's method and using these to guide the transformation of the reused artifact. Naming differences, rearrangements of lines, and different nesting of structures are largely coped with. However, since computing functional equivalence is undecidable, the correct correspondence is approximated by considering structural facts augmented with some simple semantic details, and by utilizing heuristics. The result is that ambiguities can occur where the tool cannot decide on the best option; the developer is then asked for input. The lost opportunity, at this point, is that Jigsaw can decide that a correspondence exists where the developer does not agree, and yet, they have no means to provide this feedback to adjust the integration.

DSketch. Many software change tasks involve first understanding the dependencies between different parts of the system. The typical solution involves the creation of tool support that understands the semantics of the programming language involved, to at least indicate where an entity is declared and where it is referenced. While this is well understood theoretically, developing non-trivial tooling is difficult and expensive. This cost must be amortized in savings across multiple usages of the tooling, but for domain-specific languages, minor language variants, uncommon language combinations, and specialized bridging technology, such amortization possibilities may not exist. DSketch [3] allowed the developer to create imprecise grammars for the different parts of their software, cheaply. DSketch informs the developer about matching dependencies that it has detected as a result, and the developer can modify or add grammars until they are satisfied. In the missed opportunity, the developer could have directly pointed DSketch to false/missing matches, to have it automatically adjust the grammars.

6 Conclusion

Since software engineers must deal with issues of information overload, being unable to find their way, or wanting the answer to infeasible computational problems, recommendation systems play an increasingly key role in software engineering research. For the sake of providing a simple means of understanding recent research on RSSEs, three quite different classification schemes have been outlined including a novel problem-based one for RSSEs. A few recent RSSEs have been explored, with particular focus on the domain of software reuse.

The full gamut of possibilities is seen with respect to a problem-based and a paradigm-based categorization. Some combinations of dimensions in the design-based categorization were not shown in this paper; indeed some of these do not apparently exist. A common RSSE weakness is a tendency to not utilize developer feedback to improve recommendations, and especially not for other developers. In fairness, many RSSE approaches assume external iteration in cases where the developer is not satisfied, but there are potentially significant opportunities for advancement in this dimension.

Similarly, there has been little realization of the infeasible computation problem in non-software engineering applications of recommenders. There exist cases where one would like to have a more intelligent recommender, capable of dealing with a multi-objective search problem plus a fuzzy and shifting fitness function. Opportunities abound!

Going forward, one concern is the propensity of the software engineering community to remain at an "initial evaluation" stage of validation. As recently demonstrated by Cossette and Walker [4], determining the ground truth—to know what *should* be recommended—is an expensive but crucial exercise in evaluating the performance of an RSSE; otherwise, we remain at a level of myth and opinion, which are both subject to bias and manipulation. Perhaps this is less true of other domains, but scientific progress requires hard work and careful analysis of the problems at hand, before we worry too much about "improving" putative solutions that lack solid evidence.

References

[1] Beck, K.: Test Driven Development: By Example. Addison Wesley (2002)

[2] Castro-Herrera, C., Duan, C., Cleland-Huang, J., Mobasher, B.: A recommender system for requirements elicitation in large-scale software projects. In: Proc. ACM Symp. Appl. Comput., pp. 1419–1426 (2009)

[3] Cossette, B., Walker, R.J.: DSketch: Lightweight, adaptable dependency analysis. In: Proc. ACM SIGSOFT Int. Symp. Foundations Softw. Eng., pp. 297–306 (2010)

[4] Cossette, B.E., Walker, R.J.: Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In: Proc. ACM SIGSOFT Int. Symp. Foundations Softw. Eng., pp. pp. 55/1–55/11 (2012)

[5] Cottrell, R., Walker, R.J., Denzinger, J.: Semi-automating small-scale source code reuse via structural correspondence. In: Proc. ACM SIGSOFT Int. Symp. Foundations Softw. Eng., pp. 214–225 (2008)

[6] Holmes, R., Walker, R.J.: Customized awareness: Recommending relevant external change events. In: Proc. ACM/IEEE Int. Conf. Softw. Eng., pp. 465–474 (2010)

[7] Holmes, R., Walker, R.J., Murphy, G.C.: Approximate structural context matching: An approach to recommend relevant examples. IEEE Trans. Softw. Eng. 32(12), 952–970 (2006)

[8] Hummel, O., Janjic, W., Atkinson, C.: Code Conjurer: Pulling reusable software out of thin air. IEEE Softw. 25(5), 45–52 (2008)

[9] Jannach, D., Zanker, M., Felfernig, A., Friedrich, G.: Recommender Systems: An Introduction. Cambridge University Press (2010)

[10] Lemos, O.A.L., Bajracharya, S., Ossher, J., Masiero, P.C., Lopes, C.: A test-driven approach to code search and its application to the reuse of auxiliary functionality. Inf. Softw. Technol. 53(4), 294–306 (2011)

[11] McMillan, C., Hariri, N., Poshyvanyk, D., Cleland-Huang, J., Mobasher, B.: Recommending source code for use in rapid software prototypes. In: Proc. ACM/IEEE Int. Conf. Softw. Eng., pp. 848–858 (2012)

[12] Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proc. ACM SIGSOFT Int. Symp. Foundations Softw. Eng., pp. 42/1–42/11 (2012)

[13] Reiss, S.P.: Semantics-based code search. In: Proc. ACM/IEEE Int. Conf. Softw. Eng., pp. 243–253 (2009)

[14] Robillard, M., Walker, R., Zimmermann, T.: Recommendation systems for software engineering. IEEE Softw. 27(4), 80–86 (2010)