Chapter 8
Developer Profiles for Recommendation Systems

Annie T. T. Ying and Martin P. Robillard

Abstract. Developer profiles are representations that capture the characteristics of a software developer, including software development knowledge, organizational information, and communication networks. In recommendation systems in software engineering, developer profiles can be used for personalizing recommendations and for recommending developers who can assist with a task. This chapter describes techniques for capturing, representing, storing, and using developer profiles.

8.1 Introduction

Recommendation systems in software engineering (RSSEs) seek to assist individuals in performing software engineering tasks. In many situations, useful recommendations will be independent from the developer involved in the task. For example, a recommendation to inspect a file for faults that is based on code churn metrics [37] will be the same for everyone. However, there are also situations where the relevance and quality of a recommendation will be impacted by the personal characteristics of the developer performing the task.

Consider a system that recommends elements from an application programming interface (API) that could be used to implement some functionality. A developer chooses a descriptive name for the method (e.g.,"`sortRecords()`"), enters some comments that describe its purpose, and the recommendation system outputs one or more potentially useful API elements, such as the `sort` method from a library. At first glance this sounds like a great system, until we realize that the library `sort` method gets recommended every time we implement some sorting functionality. Here the recommendation system makes the assumption that the recommended API

Annie T. T. Ying
McGill University, Montréal, Canada, e-mail: annie.ying@cs.mcgill.ca

Martin P. Robillard
McGill University, Montréal, Canada, e-mail: martin@cs.mcgill.ca

elements are unknown to the developer, who must discover them to carry out the task. For this idea to work properly, the recommendation system must be able to reason about what API elements might already be known to a developer. The functionality described above was first implemented by a system called CodeBroker [54]. One of the prominent features of CodeBroker is that it could personalize the recommendations by storing a model of the user's knowledge of an API, to avoid recommending methods known to the user.

In the context of recommendation systems, personalization is the delivery of different information (i.e., recommendations) depending on the target user [2, 13, 15, 19]. The concept of personalization is pervasive in many application domains for recommendation systems. For example: as much as 60% of Netflix's rentals result from personalized recommendations from Netflix's movie recommender [40]; and the Google search engine can provide personalized search results [16]. Outside the context of recommendation systems, more complex personalization approaches involving personalizing the delivery of the content have found success in domains such as intelligent tutoring systems [47], natural language dialog systems [51], and adaptive hypermedia [48]. Despite its successful use in commercial systems, personalization is not yet widely supported in the software engineering domain.

Customization is often considered to be a type of manual personalization. In customizable systems, users have the ability to build their own profiles by specifying preferences, typically from a list of options [15]. Systems that support customization are usually called adaptable systems. An example of an adaptable system is MyYahoo!: Users of MyYahoo! can adjust what type of content they prefer to see displayed in their homepage (e.g., type of news articles, stock prices), as well as how the content should be organized. In software engineering, the IBM Rational Application Developer provides an example of a customizable system. This system is a development environment built on top of the Eclipse integrated development environment (IDE) that allows users to specify roles, such as "Java developer" and "Web developer". These roles simplify the user interface for each role by limiting the number of available features [12].

The techniques needed to support customization within an application are well-understood. In the simplest case, customization can be implemented through a simple key–value property API. For this reason, this chapter focuses instead on the more adaptive systems that can personalize recommendations automatically, typically based on inferred user characteristics.

Beyond modeling the technical knowledge of a developer, as in the case of CodeBroker, RSSEs can take into account other characteristics of developers. These include basic information maintained by their employer (such as demographic information), but also more complex structures that capture their communication network. In the personalization community, a representation that captures these types of personal characteristics is called a *user profile* or a *user model* [2, 13, 15, 19].

In software engineering, models of developer characteristics are not only useful for personalizing recommendations: they are also the basis for producing recommendations *about* developers. For example, Expertise Recommender [32] can discover and recommend the developer who has the most expertise on a given module

by analyzing the change history of a system under development. In the case of such expert-finding tools, the developer characteristics stored and analyzed by the system under development are not necessarily those of the developer *using* the recommendation system (as in the case of personalization). For this reason, we use the term *developer profile* to refer to a collection of information about a developer, to avoid the overly-restrictive focus on users.

In this chapter, we describe a collection of techniques that can be used to build developer profiles. We begin with a discussion of the potential applications of developer profiles in software engineering, illustrated with a description of their use in three different systems (Sect. 8.2). We follow with a presentation and discussion of the techniques necessary to collect and store different types of information about developers. Section 8.3 focuses on modeling software development knowledge and Sect. 8.4 discusses organizational information and communication networks. In Sect. 8.5, we discuss general issues related to the maintenance and storage of developer profiles. We conclude in Sect. 8.6 with a short discussion of the risks and limitations of developer profiles in RSSEs.

8.2 Applications of Developer Profiles

The two main areas of application for developer profiles in RSSEs are to *personalize recommendations*, and to *recommend developers*.

8.2.1 Personalizing Recommendations

We return to CodeBroker, the RSSE introduced in Sect. 8.1, to illustrate how capturing a model of a developer's knowledge supports adaptive recommendations.

CodeBroker [54] facilitates code reuse by recommending Java methods that can be used to complete a task. Figure 8.1 shows the system operating within Emacs.

The figure shows the Java source code written by a developer in the process of implementing randomization functionality in a card game. Specifically, the developer has just finished typing in the signature of the `getRandomNumber` method, preceded by some descriptive comments. At that point the developer moves beyond the method signature (see the cursor in Fig. 8.1), and CodeBroker automatically generates recommendations (bottom view in Fig. 8.1). The top recommendation is a method named `getLong` from the `Randomizer` class. This API method generates a random number between two given long integers, essentially the functionality and signature the programmer is about to implement. Here the programmer is obviously unaware of this API method. The recommendation is useful because it saves the overhead of reimplementing `getRandomNumber`.

To produce recommendations, CodeBroker considers terms in the comments and method signature, and uses information retrieval techniques to match them with

Fig. 8.1: CodeBroker recommendations in Emacs [reproduced with permission, from 53]

methods of the Java Development Kit API. However, in this context, reuse recommendations are only useful if they support the discovery of new API methods.

To avoid generating useless and distracting recommendations for methods already known, CodeBroker maintains a developer profile that captures the API methods estimated to be known by a developer, and removes from the recommendation list any method found in the user.

A developer profile in CodeBroker initially contains the methods used by the developer. As the developer types in more code, the profile is automatically updated. Figure 8.2 shows an example of a developer profile in CodeBroker. The profile includes two methods of the `java.io.File` class that were automatically included through code analysis. In addition, the developer ("Jeff") complemented the profile by specifying that he had knowledge of the whole Java package `java.net` as well as the individual methods `getParameterInfo` and `toCharArray`. As this example shows, manual adjustments to the profile can be done at various levels of granularity (e.g., entire packages or classes).

CodeBroker uses developer profiles to provide filtering on a list of API elements recommended for reuse. The concept of information filtering is taken further in Mylyn [23], a tool that adapts the user interface of Eclipse to the present needs of a developer by hiding information (code elements) that have not been accessed

- java.applet
  - Applet
    - getParameterInfo (**added by Jeff at Thu 2 08:30:10 2000**)
- java.io
  - File
    - exists (Thu Nov 2 08:35:49 2000, Nov 2 08:15:10 2000, Nov 2 08:10:22 2000)
    - isAbsolute (Thu Nov 2 09:36:31 2000, Nov 2 09:19:15 2000)
  - CharArrayWriter
    - toCharArray (**added by Jeff at Thu 2 09:00:11 2000**)
- java.net (**added by Jeff at Thu 2 09:15:11 2000**)

Fig. 8.2: An example programmer profile in CodeBroker [53]. The top-level bullets indicates the Java package, the second level indicates the classes in the corresponding package, and the third the methods in the class

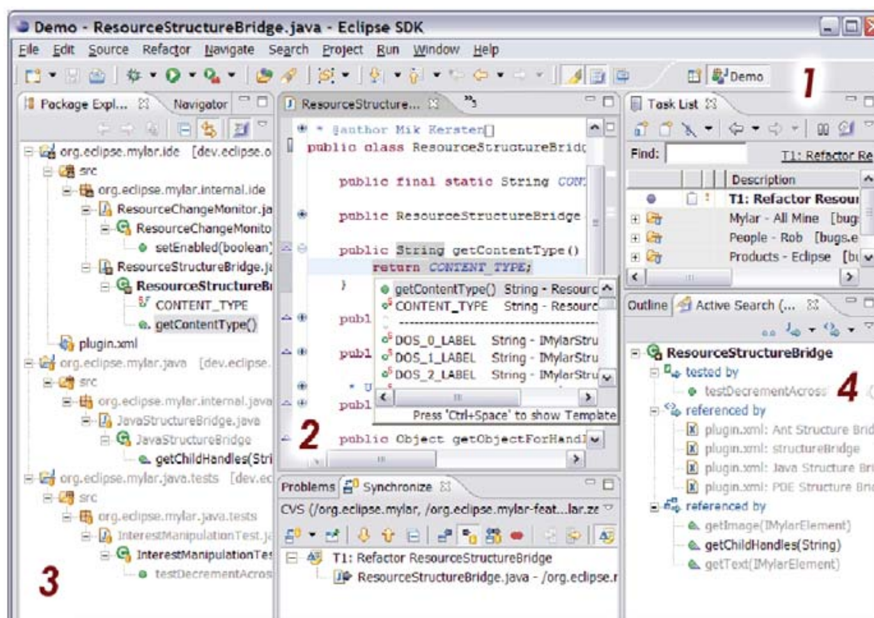

Fig. 8.3: Mylyn in action in the Eclipse IDE [reproduced with permission, from 23]

recently, and by emphasizing the parts of the user interface that are more likely to be accessed.

Figure 8.3 shows Mylyn in action as part of a scenario originally described by Kersten and Murphy [23]. Mylyn adapts various views in Eclipse (e.g., the Package Explorer: Fig. 8.3, item 3) to only show the information relevant to the current task. The task the developer is working on is entitled "Task-1: Refactor

`ResourceStructureBridge`" (marked with a solid dot in the Task List view in Fig. 8.3, item 1).

Mylyn adapts the user interface based on the developer's interaction history in the IDE. Program elements that are accessed more often and more recently as part of a task have higher importance, called degree-of-interest (DOI). For example, in a view that displays the system structure (Fig. 8.3, item 3), the only artifacts visible are the ones Mylyn estimated to be relevant. This view also marks the most relevant artifacts in bold.

In Mylyn, the structure that stores relevant elements and their corresponding DOI is called the *task context*. A user can also manually increase or decrease the DOI of the elements in a task context, resulting in direct changes to the model. We consider that task contexts are a form of developer profile that captures the immediate interest of a developer involved in a task. In addition to adapting the user interface, task contexts can also be used to restore the resources visible in the user interface at a different point in time, for example in the case of an interruption. The developer profile used in Mylyn is task-oriented and does not capture information beyond the current task.

8.2.2 Recommending Developers

RSSEs can also be used to help locate individuals with a certain expertise. The problem of identifying who has the right expertise has become increasingly important given the ubiquity of large and distributed teams [17]. For these types of recommendation systems, developer profiles constitute the data items in the knowledge base used to produce recommendations. These systems are exemplified by Expertise Recommender [32].

Expertise Recommender was designed to help technical support personnel get in contact with the people best able to solve customer support requests. Figure 8.4 shows a list of recommendations produced by Expertise Recommender. In this scenario, a technical support representative fielded a support call from a customer and entered the request "I/O Error 16 in program M.013 customer PCI" with, among others, the value "Social Network" as a filter.

In Expertise Recommender, recommendations are derived from various filtering heuristics applied to a *profile database*. In principle the profile database can capture any kind of developer information available in an organization. In the example described by McDonald and Ackerman [32], the profile database is populated from two basic data sources: the version control system (VCS) for the software, and the issue repository.

The issue repository is used to search for technical support personnel who had solved problems with the description similar to the input query. In our example scenario, the query includes three pieces of key information: the module name "M.013", the customer name "PCI", and the problem description "I/O Error 16". Each of the three pieces of information would trigger a separate search for past

Expertise Recommendation

Expertise Request:

Topic Area: Tech Support Filter: Social Network
Request: I/O Error 16 in program M.013 customer PCI

Request Results:

Susan Wright (SAW) swright@development.msc.com	Suite 103	x1297	(949) 824-5097
Keri Carpenter (KC) kerl@training.msc.com	Remote Anaheim	x1363	(714) 551-7663
Rob Klashner (ORK) klashner@support.msc.com	Suite 101	x1255	(949) 824-5055

Escalate Change Close

Fig. 8.4: An example of interface to present expertise recommendations based on Expertise Recommender [32]

problems that match the information. The recommendation is the technical support personnel who solved the past problems that best matched the three pieces of information. In addition, Expertise Recommender also supports another mode ("Change History" rather than "Tech Support") which uses source file contributions from commits and the proximity in the organization of the expert requester.

The Expertise Recommender architecture does not explicitly capture the concept of a developer profile as an explicit data structure. Instead, various optimizations are used, such as database indexes and maps. We can nevertheless consider that a developer profile in Expertise Recommender implicitly captures, in addition to basic contact information, a list of modules and a term vector that contains the important words in the description of all the issues solved by a developer. This observation illustrates the important point that developer profiles are conceptual entities that do not need to be explicitly represented as a unit in the implementation of the RSSE.

An interesting note about Expertise Recommender is that the recommendations produced take into account the characteristics of the user of the system. The "filter" parameter allows the results to be filtered according to two values: "Department" and "Social Network". The "Department" value returns developers ranked according to how close they are to the user of the system in terms of the official organizational structure (e.g., it prioritizes developers in the same department). "Social Network" instead prioritizes developers who are the closest in an ad hoc social network that takes additional personal relations into account. The developers of Expertise Recommender argue that this personalization feature helps distinguish their system as a recommendation system, in contrast to a more traditional information retrieval system. In the context of this chapter, we note that this feature makes Expertise Recommender an example of a system that uses developer profiles both as knowledge base elements and as a means to personalize the recommendations.

8.3 Development Knowledge

Software development knowledge is the knowledge developers have about both the system(s) they are working on and their general software development experience. Software development knowledge is usually derived *implicitly* rather than by explicitly asking a developer to provide it. This process requires a RSSE to infer the development knowledge from various actions performed by the developer. These actions are typically captured by three types of artifacts: change logs stored in a VCS, interaction traces collected by an IDE, and records stored in an issue tracking system. Sections 8.3.1 to 8.3.3 discuss these three data sources used for inferring development knowledge. We follow with a brief discussion of how certain types of software development knowledge can also be collected by *explicitly* asking the developer (Sect. 8.3.4). We present several common representations for software development knowledge in Sect. 8.3.5.

8.3.1 Version Control System Data

Version control system data typically refers to commits obtained from source code revision repositories, including the traditional ones such as SVN and CVS and, more recently, distributed repositories such as Git. There are several ways to use VCS data to infer development knowledge.

One heuristic is to assume that a developer changing a particular part of the source code has knowledge in that part of the code. One can find out who changed which lines of code by looking at the commit logs from a VCS. This heuristic is derived from the so-called *Line 10 rule* observed in a field study [32]: when a developer wanted to know who had the expertise for a particular part of the code, say line 10 of a particular file, the developer would consult the VCS commit log to see who was the last person changing line 10.[1] Expertise Recommender [32] uses this idea to recommend developers who have the expertise for a software module. Each developer profile includes the list of modules that a particular developer has last changed. Expertise Recommender takes a textual query as input and identifies all of the program modules mentioned. The list of recommendations are all the individuals who have modified a module mentioned in the query, ranked by recency of the last change by an individual.

The Line 10 heuristic is useful to infer knowledge about the source code being developed and when the change history of the source code is available. However, this heuristic would not work for estimating expertise about API elements, or about parts of the code for which change history is not available. For example, CodeBroker personalizes recommendations by filtering out recommendations that contain API ele-

[1] Using data from older VCSes requires mapping lines to high-level program elements such as methods. For more information about this step, readers can refer to Zimmermann and Weißgerber [58].

```
1 import p.;
2
3 class B {
4   void main() {
5     A a = new A();
6     a.p1 = "hello";
7     a.dd(a.p1);
8     a.remove(a.p1);
9   }
10 }
```

Fig. 8.5: An incomplete Java program demonstrating the difficulty to extract information about method calls [adapted from 11, p. 314]

```
package p;

public class A {
  String p1;
  void add(Object o) {}
  void remove(Object o) {}
}
```

Fig. 8.6: The part of the program missing in the incomplete program demonstrated in Fig. 8.5 [adapted from 11, p. 314]

ments known to the user. If a developer has used the Java API method `addElement` from `java.util.Vector`, CodeBroker would not recommend this API method. For this type of filtering to work, a RSSE must model the developer's knowledge of the API by analyzing what API elements are used in the code.

Extracting which API elements (e.g., methods) are used (e.g., called) from commits is a technical challenge. Identifying which method a developer is using in the code requires determining the type bindings of object variables that are the target of methods. This task is normally handled by compilers. However, in the context of a commit, resolving type bindings is technically challenging because commits from a VCS are generally a subset of the whole program, possibly without the necessary dependency information for the usual type binding resolution to work. Even when one has access to the source code and the dependencies of the whole program, it may not be practical to compile a snapshot of the whole program for every commit. Resolving type bindings in commits thus requires guessing the type bindings. One technique that can infer type bindings from commits is partial program analysis (PPA) [11].

For example, suppose that a developer added one line of source code (line 7) to class `B` in Fig. 8.5. For the purpose of building the developer profile, we want to know which method is called at line 7 so that we can store this information in the developer profile. A syntactic analysis of only the code shown in Fig. 8.5 can only

tell us that a method named `add` with one parameter is called at line 7, but not which class declares `add` nor the type of the parameter. This is especially problematic when multiple classes declare methods with the name `add` and one parameter. Improving upon pure syntactic analysis, partial program analysis infers that method `A.dd(String)` is called by looking at the string in the assignment in line 5. This inference is not strictly correct: in this example, apparently, class A (Fig. 8.6) only has one method named `add` with one parameter, `A.dd(Object)`; thus, the inference is more specific than the one provided by syntactic analysis on the full program consisting of classes `A` and `B`.

Beside the actual code location and the methods that are called at the location, additional method calls in the code lines above and below a changed line can also be considered for inclusion in a developer profile [28]. For example, if a developer changed line 7 in the code in Fig. 8.5, we could also infer that the developer also knows the method `remove` at line 8. Textual terms extracted from commit messages and the actual commit have been used as a surrogate of expertise for the purpose of bug triage [31]. Terms extracted from the commit are taken from the identifiers and the comments of the code. Bug triage is a process necessary in many open source projects that maintain an issue tracking system. This triage process can become a resource-intensive task when a significant number of incoming bug reports are filed by outsiders because a project member has to determine whether an incoming bug report is valid and, if so, who should be assigned to the task of fixing the bug.

8.3.2 Interaction History Data

The second type of data one can mine to capture a developer's technical knowledge is interaction history. Interaction history refers to a sequence of events initiated by actions performed by a user with a tool. In Chap. 7, Maalej et al. [29] provide a detailed discussion on interaction history. We show how a RSSE adapts its output to individual users by looking at Mylyn [23]. Mylyn adapts various views in Eclipse to only show the information relevant to the current task. Mylyn harnesses interaction history involving software artifacts recorded in an IDE as a surrogate of relevance to the task.

In Mylyn, some events are the direct result of a developer's interaction with program artifacts. These events include *selections* (such as selecting a Java method and viewing its source) and *edits*. Other events are caused by indirect interactions. For example, when refactoring a class `ResourceStructureBridge` to a different name in Eclipse, Eclipse will update the name of the classes referencing the renamed class. Each of the referencing class results in an indirect event called *propagation*. As part of this refactoring, Mylyn also tracks the actual rename operation provided by Eclipse, called a *command* event in Mylyn. Table 8.1 shows the events corresponding to this refactoring operation, as part of the task "Task-1: Refactor `ResourceStructureBridge`" first described in Sect. 8.2. The columns with a name prefixed with "Event" denote pieces of information captured in a Mylyn event and

Developer action	Event #	Event kind	Event origin	Event target(s)
select `RSB`	1	selection	Package Explorer	class
rename `RSB`	2–5	propagation	Package Explorer	source file, package source folder, project
rename `RSB`	6	command	Rename refactoring	class

Table 8.1: Sample events in the interaction history captured by Mylyn [adapted from 23]. "`RSB`" is short for "`ResourceStructureBridge`"

the column "Developer action" describes the event. For simplicity, we use an event number instead of the timestamp of an event (the column named "Event #"). The column "Event origin" refers to the tool associated with the event recorded and the column "Event target(s)" refers to the software artifacts associated with the event. Event 1 corresponds to the developer selecting the class. Events 2-6 correspond to the propagation and command events resulting from the rename operation.

Another example of an indirect event is when the developer selects the `get ContentType` method (Fig. 8.3, item 2). For each structural parent of the method (its class, source file, package, source folder, and project), Mylyn creates a propagation event. These propagation events cause the structural parents to become relevant and, therefore, visible in the Package Explorer (Fig. 8.3, item 3).

In addition to changes and navigation to code elements, a wide range of other interactions can be observed in an IDE. In the web domain, researchers have found that linger time and amount of scrolling can be useful indicators of interest [2]. Evidence to this effect is mixed in the software engineering domain [1, 44]: Mylyn [23] uses how frequently and recently a program element is being accessed, but not how long a developer stays on a program element nor how much a developer scroll within a program element, as its means for retaining source files visited by a user as relevant for the task. A study found that scrolling does not indicate interest or importance of the element, but rather an indication that the developer is lost [44].

Navigating to a code element and checking in changes to a VCS can imply different levels of familiarity on the source code. Fritz et al. [14] found that initial authorship of a code element is the strongest factor for predicting the correct level of source code familiarity (compared to subsequent authorship, navigating to the code element, and intermediate editing of the code element). Robbes and Lanza [43] compared algorithms making use of various types of implicit data, interaction history and commits, in terms of their predictability of the next change.

8.3.3 Issue Tracking System Data

The third type of data source that can reveal development knowledge is issue tracking systems. Issue tracking systems refer to systems that maintain lists of issues such as software bugs. In Chap. 6, Herzig and Zeller [18] provide some practical advice in mining bug reports. In open source projects, issue tracking systems allow users to report bugs directly to the open source developers. Many commercial software organizations use issue tracking systems as an internal medium for coordinating software testers, developers, and managers in reporting, prioritizing, and discussing issues. Another type of issue tracking systems keeps track of technical support call tickets in a call center.

Expertise Recommender [32] identifies technical support personnel who can resolve a customer support request. The design was motivated by a field study [32] on the process for identifying which of the technical support staff can solve a technical call. The heuristic is to find similar technical calls completed in the past, by first querying the support database and then determining which of the results were similar to the current problem.

For each technical support staff member, Expertise Recommender builds an entry in the profile database using three pieces of information from the technical support problems resolved by the given person: the problem description, the customer, and the module responsible for the problem. These three fields are used to build three term vectors that characterize a staff member. A term vector's dimensions are textual terms, and the value of a dimension is the number of times a term appears in the past problems resolved by the staff member. In Chap. 3, Menzies [33] explains the concept of term vectors in more detail.

In the scenario described in Sect. 8.2, the customer "PCI" complained in the call that the system has a file error "I/O Error 16" in the module called "M.013". The representative taking the call was familiar with the module in general, but was unsure why this particular customer was experiencing the file error. For this representative's profile, the customer vector's "PCI" dimension would have value 0 because the representative had not dealt with the customer "PCI" before, while the module vectors "M.013" dimension would have a positive value because the representative had dealt with the module in previous problems. On the other hand, the top personnel recommended in Fig. 8.4, Susan Wright, most likely had positive values for the customer vector's dimension for "PCI", the module vector's dimension for "M.013", and the problem description's dimension for "I/O Error 16".

In Expertise Recommender, a term vector is normalized using a master term vector which represents the total number of times a term is used in the entire problem database. Other technical challenges in dealing with textual terms include building a thesaurus and handling misspellings and abbreviations.

8.3.4 Explicit Data Collection

The data collection strategies presented above estimate development knowledge using heuristics applied to various data sources. A more direct way to obtain development knowledge is to explicitly ask the developer to provide the information. In e-commerce, data provided explicitly usually comes in the form of user ratings. A classical example is the Netflix movie recommendation system. For the Netflix system to provide useful recommendations, users have to first explicitly provide examples of movies they like or dislike in a scale of one to five stars.

Besides the fact that explicit data collection imposes a burden on the user and may not scale in many situations, a problem with manually generated profiles is that users may not have the ability to evaluate their own expertise. Because of these two problems, it may not be practical for RSSEs to ask a developer to provide the level of expertise for each individual item.

Instead of solely depending on explicit information, RSSEs can elicit information from the user to complement the implicitly-captured information. To reduce the effort from the user, explicit information can be collected at a *coarse-grained* level, where a user indicates a large group of items (in contrast to a fine-grained approach, where a user provide information for individual items). CodeBroker, described in Sect. 8.2, allows a user to manually adjust the developer profile. Developers can do so by specifying that they have knowledge at a coarse-grained level, for example, on a whole Java package `java.net` in the developer profile demonstrated in Fig. 8.2.

In terms of the quality of data obtained explicitly versus implicitly, in the web domain, conclusions have shifted over time. Earlier opinions suggested that implicitly-collected data was of lower quality than explicitly collected data [21]. More recent studies provide a more positive perspective on the usefulness of implicitly-collected data [49, 52]. There is not as much investigation in software engineering. Fritz et al. [14] investigated two types of implicitly collected information—interaction history and commits from a VCS—to construct a model a developer's familiarity with a given code element. The study shows that data from the commits from VCS is a better indicator than interaction history when inferring a developer's familiarity to a code element.

8.3.5 Representation

The simplest way to represent *what* a developer knows is to list the signature of the program elements a developer has knowledge of, without distinguishing the *extent* of the knowledge. For example, CodeBroker's user profile (Fig. 8.2) includes the list of method signature for all of the methods in a developer profile, organized in terms of enclosing class and package. CodeBroker does not model the extent of the knowledge but implicitly assumes that a developer has the same knowledge on every method listed in the developer profile.

This assumption is not always appropriate. In Expertise Browser, a tool analogous to Expertise Recommender, the developer profile is also a list of what is called *experience atoms*: locations of source code checked in by a developer [35]. Expertise Browser keep counts of how many times a particular line of source code has been modified by a developer. These experience atoms can be used to reason about the expertise of a person, or aggregated to reason about the expertise of an organization. These counts can also be used to rank recommendations for the most expert developer on a given part of the code.

The assumption of Expertise Browser is that each use of a program element increases the extent of the knowledge equally. However, different weighting schemes can also be considered for this purpose. For example, the frequency counts can be normalized by the global counts computed on all individuals. The intuition is that common methods such as `List.dd` would get less weight because they are used many times by many individuals, whereas rarer methods would get a higher weight. The extent of the knowledge can also be affected by how recently the developer acquired the knowledge. This idea is discussed in the profile maintenance section (Sect. 8.5).

A developer profile is not limited to the program elements that a developer interacted with directly, but can also represent *relationships* between program elements. In Mylyn, indirect events such as the propagation event in Table 8.1 represent relationships among program elements. However, Mylyn does not store these relationships in a persistent task context. An in-memory task context graph representing these program elements and relationships is constructed by processing the events in the persisted task context. The benefit of this approach is that the task context can be shared with other developers. When loaded on another developer's workspace, the graph can be adjusted to adapt to the program elements and relationships present in the other developer's workspace. When reconstructing the in-memory representation on another developer's workspace, additional interaction history incurred by the other developer can also augment the representation.

The weighting for each element in the task context, called degree-of-interest (DOI), is derived from the frequency and recency of the events in the interaction history. The frequency is the number of interaction events that refer to the element as a target. Each type of event has a different scaling factor, resulting in different weightings for different kinds of interactions. Old events are weighted less because of a decay function, as discussed in Sect. 8.5. The DOI of a relation, consisting of a source and a target element, is computed using the same DOI algorithm, by means of the relation's target element.

Instead of capturing a list of program elements, it is also possible to aggregate the knowledge they represent by using term vectors that represent a normalized version of the frequency of individual terms used in software development artifacts authored or changed by a developer [31]. Vector-based user profiles thus refer to the representation popular in information retrieval for textual documents: a document is represented as a vector of n dimensions, where the dimensions correspond to n in-

dexed textual terms[2] in the corpus of documents. For example, instead of containing a list of methods a developer has used, a developer profile could contain a method-by-term matrix where every row represents a method and every column, a term in the vocabulary of all terms in the signatures of methods in a program (assuming method identifiers are tokenized, e.g., by relying on the camel case convention). It is also possible for the term vectors to capture information in the source code of the method, their comments, etc. In Chap. 3, Menzies [33] dives deeper into information retrieval.

This approach is used by a bug triage recommendation system where a user profile is the text extracted from the most recently fixed bug reports [4]. Each vector in the profile contains text converted from free-form text in the summary and description of a report. A value in the vector indicates the frequency of a particular term, normalized by the length of the bug report, total intra-document frequency, and inter-document frequency. Another bug triage recommendation system also used a vector representation, not to model the text from bug report but terms extracted from the source code changes by a given developer [31]. A value in a vector is the term frequency present in a source code change.

More sophisticated representations and algorithms have been explored in domains outside software engineering, for example, in adaptive educational systems and personalized information retrieval. A survey by Steichen et al. [48] provides an overview.

8.4 Organizational Information and Communication Networks

To make recommendations in a software engineering context, it is also possible to leverage data about the position of developers in their organization, or information about their communication networks. Organizational information includes characteristics such as a developer's position in the organization, the management structure, and a developer's role in the software development process. Developers' communication networks are the graphs that models their various interactions with other people for work purposes.

Organizational information and communication networks can play various roles in RSSEs, from being the central data source used to generate recommendation to serving as a filter for recommendations generated in some other way. For example, organizational information can be used as the basis for predicting fault-prone modules [38], Expertise Browser allows users to explore recommended experts by organizational structure [35], and Expertise Recommender offers the capability to filter its recommendations to only show those in the social network [32].

[2] Terms are typically tokens from document corpus, stemmed or not depending on the application, after removing stop words and non-alphabetic tokens [30].

8.4.1 Data Collection

Organizational information can normally be obtained from sources such as company organization directories and software project management servers. A standard protocol for accessing and maintaining a company organizational directory is the Lightweight Directory Access Protocol (LDAP). Such a directory typically contains the geographical location and the position of an employee. Many company directories are built on a LDAP server as the back-end and use the LDAP's query facility. A example query that such company directories support is to find all employees located in a particular city and have a particular role (e.g., "software developer"). RSSEs designers can use the query facility to obtain organizational information.

Software development roles can be implicitly mined from an integrated software development management platform, such as IBM Rational Team Concert [9]. Team Concert allows a software development team to create and specify roles. This information can be obtained from the Java API of Team Concert.

Developers' communication networks can be inferred directly from communication media such as emails and IRC chat logs, for example by drawing edges between senders and receivers. Communication networks can also be built from a variety of other sources, depending on what can be considered to constitute evidence of communication. Sources such as VCSes and issue tracking systems can be used as an indicator of interaction by considering that developers are related if they have been involved with the same issue. Involvement can be specified as a subset of any of the possible ways for a developer to be associated with a bug report, including fixing the bug described in the issue report, being CCed ("carbon-copied") on emails related to the issue, or providing comments on the issue [3].

Two important technical challenges when building communication networks from VCS data and emails are *linking* bug reports with the source code artifacts that solve them, and *de-aliasing* different email addresses that may be associated with the same person. Unless sufficient care is taken to control imprecision in the linking and email aliases, these issues can introduce a significant amount of noise in the resulting networks. Bird et al. [6] offer an in-depth discussion of the problem of linking bug reports to the corresponding fixes [5]. For email de-aliasing, one algorithm found to work well is to group email addresses using a clustering algorithm. (In Chap. 6, Herzig and Zeller [18] provide more information about clustering algorithms in general.) The clustering algorithm requires a similarity function that returns a similarity value between any pair of email addresses. The output of the algorithm are groups of email addresses; the email addresses in a group are predicted to be associated with the same individual.

STeP_IN is an example of RSSE that relies on organizational data and communication networks [55]. STeP_IN recommends relevant experts and artifacts based on information obtained from VCSes, issue tracking systems, and a communication network derived from email archives. One unique feature of STeP_IN is that instead of simply capturing communication relationships, STeP_IN also models how likely it is that the person being recommended wishes to be involved in the communication. Specifically, STeP_IN captures the concept of *obligation*, to avoid one con-

$$\begin{array}{c} \\ \begin{array}{c|ccccc|} & f_1 & f_2 & f_3 & f_4 & f_5 \\ d_1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ d_2 & \cdot & \cdot & \cdot & \cdot & \cdot \\ d_3 & \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \end{array} \begin{array}{c|ccccc|} & f_1 & f_2 & f_3 & f_4 & f_5 \\ f_1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_2 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_3 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_4 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_5 & \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \begin{array}{c|ccc|} & d_1 & d_2 & d_3 \\ f_1 & \cdot & \cdot & \cdot \\ f_2 & \cdot & \cdot & \cdot \\ f_3 & \cdot & \cdot & \cdot \\ f_4 & \cdot & \cdot & \cdot \\ f_5 & \cdot & \cdot & \cdot \end{array} = \begin{array}{c|ccc|} & d_1 & d_2 & d_3 \\ d_1 & \cdot & \cdot & \cdot \\ d_2 & \cdot & \cdot & \cdot \\ d_3 & \cdot & \cdot & \cdot \end{array}$$

Fig. 8.7
Dimensions of the input matrices and the final matrix representing coordination requirements

stantly asking help from a particular colleague, or to avoid one only asking help but never helping others. In STeP_IN, a user can explicitly change the preference value describing whether to be involved in an interaction with a particular colleague.

8.4.2 Representation

A communication network is a graph that can simply be represented as an $n \times n$ adjacency matrix, where n is the number of nodes in the graph and a value of 1 in a cell ij represents an arc between nodes i and j (and 0 otherwise).

Coordination requirements illustrate how a matrix representation of developer characteristics can be employed to compute derived information about a developers. Coordination requirements represent a type of recommendation, namely, information about who a developer should coordinate with to best complete their work [8].

Conceptually, the approach uses two input matrices:

1. a *file authorship matrix* (a developer by file matrix), where a cell ij indicates the number of times a developer i has committed to a file j; and
2. a *file dependency matrix* (a file by file matrix), where a cell ij (or ji) indicates the number of times the files i and j have been committed together.

The approach computes coordination requirement through two matrix multiplications on three input matrices (Fig. 8.7). The first product multiplies the file authorship matrix and the file dependency matrix, resulting in a developer by file matrix. This matrix represents the set of files a developer should be aware of given the files the developer has committed and the relationships of those files with other files in the system. To obtain a representation of coordination requirements (a developer by developer matrix), the approach then multiplies the first product with the transpose of the file authorship matrix. This final product is a matrix where a cell ij (or ji) represents the amount of shared expertise of developers i and j. More precisely, the matrix describes the extent to which developer i committed files that share commit-relationships with files committed by developer j.

Emergent Expertise Locator is a recommendation system that builds on the concept of coordination requirements [34]. To construct a profile specific to a given developer d_1, Emergent Expertise Locator constructs the coordination requirements

$$
\begin{array}{c}
\begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline d_1 & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}
\end{array}
\;
\begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline f_1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_2 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_3 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_4 & \cdot & \cdot & \cdot & \cdot & \cdot \\ f_5 & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}
\;
\begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline f_1 & \cdot & \cdot & \cdot \\ f_2 & \cdot & \cdot & \cdot \\ f_3 & \cdot & \cdot & \cdot \\ f_4 & \cdot & \cdot & \cdot \\ f_5 & \cdot & \cdot & \cdot \end{array}
=
\begin{array}{c|ccc} & d_1 & d_2 & d_3 \\ \hline d_1 & \cdot & \cdot & \cdot \end{array}
$$

Fig. 8.8: Dimensions of the input and the coordination requirements for a given developer d_1

on the fly, focusing on the current developer. As a result, the product is a vector that represents the coordination requirements relevant to the given developer (see Fig. 8.8).

8.5 Profile Maintenance and Storage

A number of design decisions can impact developer profiles and their use in RSSEs. This section discusses two important design dimensions for RSSEs using developer profiles: profile maintenance, and profile storage. Profile maintenance concerns whether a RSSE can adapt a developer profile over time. This issue is referred as *adaptivity* by the user modeling and recommendation system community [20, 36]. Profile storage is concerned with which component of a RSSE the profile is constructed and stored.

8.5.1 Adapting Developer Profiles

The simplest approach to profile maintenance is to keep the profile static. When using implicitly gathered data, this means that the user profile is based on the entire available data at the time of the profile construction such as. For example, usage expertise can be mined from the entire change history of a system [45]. However, in any high-churn situation, static user profiles will quickly get out of date. There are two ways to improve upon static profiles.

First, in a system that builds developer profiles in a batch mode, the profiles can be manually adapted by requiring the user to specify which time period corresponds to the current experience [35]. In such cases we would consider the profiles to be *adaptable*.

Second, in more dynamic systems, another strategy is to ask the user to set some parameters that guide the adaptation of the user profile. In Mylyn [23] for example, a developer needs to explicitly declare the current task, so that the Mylyn monitor can identify the boundary of the interaction history that belongs to the current task context. This design decision is a result from the user study performed on an earlier

version of Mylyn [22]: that version of Mylyn did not have the notion of tasks. The user profile was built from a single stream of interaction history, where the relative importance of older interaction history decayed automatically.

The motivation for asking a programmer to explicitly declare which task the programmer is working on is that programmers tend to switch between multiple tasks. This semi-automated approach to user profiling helps to make the profiles partly adaptive. Some work has been proposed to support the identification of task boundaries. The SpyWare tool displays a visualization and identifies sessions of work based on several measures including the number of edits per minute [42]. Coman and Sillitti [10] proposed an approach to segment development sessions.

The problem of automatically detecting parts of the interaction history that belongs to a task remains a hard problem [53], but its solution would eventually make user profiles completely adaptive. The following section presents additional strategies for achieving adaptive developer profiles.

The most straightforward way for a RSSE to adapt a developer profile is through a fixed time window. Here the notion of time can be defined either as the usual elapsed clock time [4], but also in terms of a fixed number of events in the interaction history [46, 50].

Different strategies are possible for eliminating data outside a time window of interest. The simplest is obviously to delete older events. However, when developer profiles associate data with a degree of association (in contrast to a binary, in-or-out, model of what pieces of information are associated with a developer), it is also possible to decay the association of older elements. For example, Mylyn uses a decay function for program elements in a task context. In Mylyn, the decay is proportional to the total number of events associated with the task. As another example, Matter et al. [31] employed a 3% weekly decay on VCS commits used for building a developer profile; this decay provides the optimal level of accuracy in the bug triage predictions.

8.5.2 Storing Developer Profiles

The major design decisions for storing developer profiles is whether to store them on server components (e.g., in the back-end tier in a multi-tier architecture), or in the client component used directly by users.

Developer profiles based on information mined from server repositories tend to be stored on servers. Such repositories include VCS and issue tracking systems as discussed in Sect. 8.3. Systems that employ profiles based on organizational information and communication networks need information about multiple users [8, 17, 39]. Such systems typically require a server-based approach. The STeP_IN system models a software project as a server-based project memory with relations between artifacts and their socio-technical links with developers [56].

A major concern with the collection and storage of developer information in a server is privacy. In RSSEs, privacy is a concern especially when data is collected

implicitly (for example, interaction history) and stored on a server. A simple solution is to allow a user to disable the data collection. For example Mylyn, which monitors a developer's interaction history, has a "silent activity mode" [22]. However, since tools like Mylyn base their recommendations on interaction history, disabling the collection of interaction history completely renders the tool useless. RSSE designers interested in other ways to respect privacy can consult research [e.g., 7, 24–26] on privacy-preserving personalization and recommendation systems.

In storing interaction history on the server, Mylyn [23] is somewhat of an exception. Typically, interaction history is stored in the client side, for privacy reasons but also because of the voluminous nature of raw interaction history. Conceptually, the interaction history is a sequence of ordered events. If storing interaction history on the server is important, data compression strategies must be considered. For example, Mylyn does not record all user actions. Most of the events involving the same program element used the same way are aggregated. When such an aggregation happens, the event data stores two timestamps instead of one: the timestamp of the first event and the timestamp of the last event being aggregated. Mylyn stores task contexts offline as a compact representation of the interaction history in an XML file in the client side [23]. Such an XML file is designed to be uploaded with the corresponding task, a bug report, or a feature request, if the user chooses to share a task context. One advantage of this client-based approach is the portability of task contexts as they can be used by other tools and analyses [27, 57].

Systems that support customization (see Sect. 8.1) usually employ developer profiles on the client side. In the web domain, websites such as MyYahoo! store customization information in cookies, pieces of data sent from a website and stored in the user's web browser [20]. Analogously, in software engineering, UI customization information is typically stored as an individual's local settings.

8.6 Conclusion

In our daily interaction, many of us have already been immersed in adaptive recommendation systems as we browse results from a search engine, choose a book to purchase online, or decide on a movie. These systems implement a type of personalization. Adaptive recommendations are constructed based on a user's interest represented by a user model. In the context of RSSEs, developer profiles support not only the adaptive recommendations *to* users, but also the ability to generate recommendation *about* developers.

In this chapter, we reviewed the techniques necessary for constructing developer profiles employed by adaptive recommendation systems for software engineering. Developer profiles can capture a wide range of characteristics about developers including their development knowledge, organizational information, and communication networks.

Many of the issues discussed in this chapter overlap with other aspects of RSSEs. VCS data, interaction history, and issue tracking systems are the key data sources for

generating developer profiles, but they are also used for generating many different types of recommendationsand 5 . Designing developer profiles that accurately and reliably capture the true characteristics of a developer is an empirical endeavour that will require much experimentation. The concept of personalization is also intimately tied with usability issues. Readers interested in the general area of personalization can consult several surveys [2, 15, 19, 20, 36, 48].

Even though personalization can be effective in supporting developers in their information acquisition tasks, there are concerns that adaptive systems can be too personal, up to a point where individuals are segregated into information silos. By not making available information that is available to others [40]. In the context of software engineering, a developer may discover information that is irrelevant to the current task but may increase the developer's overall knowledge and appreciation of the project. Recommendation systems that focus developers' information discovery too narrowly may negatively impact the developer's overall performance, even if they successfully support them for individual tasks. An RSSE recommending only relevant parts of the code to examine for the current task will not allow such serendipitous opportunities beneficial beyond the current task. Similarly, an expert-finding tool cannot provide all the information that would be gathered through impromptu water-cooler conversations. From the technical point of view, Ricci et al. [41] suggest to use active learning, which "allows the system to actively influence the items the user is exposed to [...], as well as by enabling the user to explore his/her interests freely".

When we consider the individual developer receiving recommendations from an RSSEs, they inevitably have differences in experience, ability, and needs. In cases where a recommendation depends on the individual receiving the recommendation, using developer profiles in RSSEs should then contribute towards improving the quality of recommendations, ideally without stifling the developer's freedom to explore and discover.

Acknowledgements

We are grateful for the help from the following people and organizations: Christoph Treude helped us greatly improve the structure of the chapter since early on and provided us comments on a previous draft. Ben Steichen acted as a reviewer external to software engineering, provided us with his expert advice on user modeling, and gave us numerous pointers to work in the user modeling community. The editors of this book provided guidance and feedback throughout the whole writing process. Mik Kersten and Yunwen Ye kindly allowed us to reproduce figures from their respective theses. Finally, NSERC and McGill have provided financial support through a number of scholarships.

References

1. de Alwis, B., Murphy, G.C.: Using visual momentum to explain disorientation in the Eclipse IDE. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 51–54 (2006). DOI 10.1109/VLHCC.2006.49
2. Anand, S., Mobasher, B.: Intelligent techniques for web personalization. In: Revised Selected Papers of the IJCAI Workshop on Intelligent Techniques for Web Personalization, *Lecture Notes in Computer Science*, vol. 3169, pp. 1–36 (2005). DOI 10.1007/11577935_1
3. Anvik, J., Murphy, G.C.: Determining implementation expertise from bug reports. In: Proceedings of the International Workshop on Mining Software Repositories (2007). DOI 10.1109/MSR.2007.7
4. Anvik, J., Murphy, G.C.: Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Transactions on Software Engineering and Methodology **20**(3), 10:1–10:35 (2011). DOI 10.1145/2000791.2000794
5. Bachmann, A., Bird, C., Rahman, F., Devanbu, P., Bernstein, A.: The missing links: Bugs and bug-fix commits. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 97–106 (2010). DOI 10.1145/1882291.1882308
6. Bird, C., Gourley, A., Devanbu, P., Gertz, M., Swaminathan, A.: Mining email social networks. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 137–143 (2006). DOI 10.1145/1137983.1138016
7. Canny, J.: Collaborative filtering with privacy. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 45–57 (2002). DOI 10.1109/SECPRI.2002.1004361
8. Cataldo, M., Wagstrom, P.A., Herbsleb, J.D., Carley, K.M.: Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 353–362 (2006). DOI 10.1145/1180875.1180929
9. Cheng, L.T., de Souza, C.R.B., Hupfer, S., Patterson, J., Ross, S.: Building collaboration into IDEs. ACM Queue **1**(9), 40–50 (2003). DOI 10.1145/966789.966803
10. Coman, I.D., Sillitti, A.: Automated identification of tasks in development sessions. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 212–217 (2008). DOI 10.1109/ICPC.2008.16
11. Dagenais, B., Hendren, L.: Enabling static analysis for partial Java programs. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 313–328 (2008). DOI 10.1145/1449955.1449790
12. Findlater, L., McGrenere, J., Modjeska, D.: Evaluation of a role-based approach for customizing a complex development environment. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 1267–1270 (2008). DOI 10.1145/1357054.1357251
13. Fischer, G.: User modeling in human–computer interaction. User Modeling and User-Adapted Interaction **11**(1), 65–86 (2001). DOI 10.1023/A:1011145532042
14. Fritz, T., Ou, J., Murphy, G.C., Murphy-Hill, E.: A degree-of-knowledge model to capture source code familiarity. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 385–394 (2010). DOI 10.1145/1806799.1806856
15. Gauch, S., Speretta, M., Chandramouli, A., Micarelli, A.: User profiles for personalized information access. In: Brusilovsky, P., Kobsa, A., Nejdl, W. (eds.) The Adaptive Web: Methods and Strategies of Web Personalization, *Lecture Notes in Computer Science*, vol. 4321, Chap. 2, pp. 54–89. Springer (2007). DOI 10.1007/978-3-540-72079-9_2
16. Google Official Blog: Personalized search for everyone (2009). URL http://googleblog.blogspot.de/2009/12/personalized-search-for-everyone.html. [retrieved 9 October 2013]
17. Herbsleb, J.D.: Global software engineering: The future of socio-technical coordination. In: Proceedings of the Future of Software Engineering, pp. 188–198 (2007). DOI 10.1109/FOSE.2007.5

18. Herzig, K., Zeller, A.: Mining bug data: A practitioner's guide. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 6. Springer (2014)
19. Jameson, A.: Adaptive interfaces and agents. In: Sears, A., Jacko, J.A. (eds.) The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications, 2nd edn., pp. 433–458. CRC Press (2008)
20. Keenoy, K., Levene, M.: Personalisation of web search. In: Proceedings of the IJCAI Workshop on Intelligent Techniques for Web Personalization, *Lecture Notes in Computer Science*, vol. 3169, pp. 201–228 (2005). DOI 10.1007/11577935_11
21. Kelly, D., Teevan, J.: Implicit feedback for inferring user preference: A bibliography. ACM SIGIR Forum **37**(2), 18–28 (2003). DOI 10.1145/959258.959260
22. Kersten, M., Murphy, G.C.: Mylar: A degree-of-interest model for IDEs. In: Proceedings of the International Conference on Aspect-Oriented Software Deveopment, pp. 159–168 (2005). DOI 10.1145/1052898.1052912
23. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1–11 (2006). DOI 10.1145/1181775.1181777
24. Kobsa, A.: Privacy-enhanced personalization. Communications of the ACM **50**(8), 24–33 (2007). DOI 10.1145/1278201.1278202
25. Kobsa, A.: Privacy-enhanced web personalization. In: Brusilovsky, P., Kobsa, A., Nejdl, W. (eds.) The Adaptive Web: Methods and Strategies of Web Personalization, Chap. 21, pp. 628–670. Springer (2007). DOI 10.1007/978-3-540-72079-9_21
26. Lam, S.K.T., Frankowski, D., Riedl, J.: Do you trust your recommendations?: An exploration of security and privacy issues in recommender systems. In: Proceedings of the International Conference on Emerging Trends in Information and Communication Security, *Lecture Notes in Computer Science*, vol. 3995, pp. 14–29 (2006). DOI 10.1007/11766155_2
27. Lee, T., Nam, J., Han, D., Kim, S., In, H.P.: Micro interaction metrics for defect prediction. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 311–321 (2011). DOI 10.1145/2025113.2025156
28. Ma, D., Schuler, D., Zimmermann, T., Sillito, J.: Expert recommendation with usage expertise. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 535–538 (2009). DOI 10.1109/ICSM.2009.5306386
29. Maalej, W., Fritz, T., Robbes, R.: Collecting and processing interaction data for recommendation systems. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 7. Springer (2014)
30. Manning, C.D., Raghavan, P., Schutze, H.: Introduction to Information Retrieval. Cambridge University Press (2008)
31. Matter, D., Kuhn, A., Nierstrasz, O.: Assigning bug reports using a vocabulary-based expertise model of developers. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 131–140 (2009). DOI 10.1109/MSR.2009.5069491
32. McDonald, D.W., Ackerman, M.S.: Expertise Recommender: A flexible recommendation system and architecture. In: Proceedings of the ACM Conference on Computer Supported Cooperative Work, pp. 231–240 (2000). DOI 10.1145/358916.358994
33. Menzies, T.: Data mining: A tutorial. In: Robillard, M., Maalej, W., Walker, R.J., Zimmermann, T. (eds.) Recommendation Systems in Software Engineering, Chap. 3. Springer (2014)
34. Minto, S., Murphy, G.C.: Recommending emergent teams. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 5:1–5:8 (2007). DOI 10.1109/MSR.2007.27
35. Mockus, A., Herbsleb, J.D.: Expertise Browser: A quantitative approach to identifying expertise. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 503–512 (2002). DOI 10.1145/581339.581401
36. Montaner, M., López, B., De La Rosa, J.L.: A taxonomy of recommender agents on the internet. Artifical Intelligence Review **19**(4), 285–330 (2003). DOI 10.1023/A:1022850703159

37. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 284–292 (2005). DOI 10.1145/1062455.1062514
38. Nagappan, N., Murphy, B., Basili, V.: The influence of organizational structure on software quality: An empirical case study. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 521–530 (2008). DOI 10.1145/1368088.1368160
39. Ohira, M., Ohsugi, N., Ohoka, T., Matsumoto, K.: Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 15:1–15:5 (2005). DOI 10.1145/1083142.1083163
40. Pariser, E.: The Filter bubble: What the Internet Is Hiding from You. Penguin Press HC (2011)
41. Ricci, F., Rokach, L., Shapira, B.: Introduction to Recommender Systems Handbook. In: Ricci, F., Rokach, L., Shapira, B. (eds.) Recommender Systems Handbook, pp. 1–35. Springer (2011). DOI 10.1007/978-0-387-85820-3_1
42. Robbes, R., Lanza, M.: Characterizing and understanding development sessions. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 155–166 (2007). DOI 10.1109/ICPC.2007.12
43. Robbes, R., Lanza, M.: Improving code completion with program history. Automated Software Engineering: An International Journal **17**(2), 181–212 (2010). DOI 10.1007/s10515-010-0064-x
44. Robillard, M.P., Coelho, W., Murphy, G.C.: How effective developers investigate source code: An exploratory study. IEEE Transactions on Software Engineering **30**(12), 889–903 (2004). DOI 10.1109/TSE.2004.101
45. Schuler, D., Zimmermann, T.: Mining usage expertise from version archives. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 121–124 (2008). DOI 10.1145/1370750.1370779
46. Singer, J., Elves, R., Storey, M.A.D.: NavTracks: Supporting navigation in software maintenance. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 325–334 (2005). DOI 10.1109/ICSM.2005.66
47. Sleeman, D., Brown, J.S.: Intelligent Tutoring Systems. Academic Press (1982)
48. Steichen, B., Ashman, H., Wade, V.: A comparative survey of Personalised Information Retrieval and Adaptive Hypermedia techniques. Information Processing and Management **48**(4), 698–724 (2012). DOI 10.1016/j.ipm.2011.12.004
49. Teevan, J., Dumais, S.T., Horvitz, E.: Personalizing search via automated analysis of interests and activities. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 449–456 (2005). DOI 10.1145/1076034.1076111
50. Viriyakattiyaporn, P., Murphy, G.C.: Improving program navigation with an active help system. In: Proceedings of the IBM Centre for Advanced Studies Conference on Collaborative Research, pp. 27–41 (2010). DOI 10.1145/1923947.1923951
51. Wahlster, W., Kobsa, A.: User models in dialog systems. In: Kobsa, A., Wahlster, W. (eds.) User Models in Dialog Systems, Symbolic Computation, Chap. 1, pp. 4–34. Springer (1989). DOI 10.1007/978-3-642-83230-7_1
52. White, R.W., Ruthven, I., Jose, J.M.: Finding relevant documents using top ranking sentences: An evaluation of two alternative schemes. In: Proceedings of the ACM SIGIR International Conference on Research and Development in Information Retrieval, pp. 57–64 (2002). DOI 10.1145/564376.564389
53. Ye, Y.: Supporting component-based software development with active component repository systems. Ph.D. thesis, Department of Computer Science, University of Colorado, Boulder (2001)
54. Ye, Y., Fischer, G.: Supporting reuse by delivering task-relevant and personalized information. In: Proceedings of the ACM/IEEE International Conference on Software Engineering, pp. 513–523 (2002). DOI 10.1145/581339.581402
55. Ye, Y., Yamamoto, Y., Nakakoji, K.: A socio-technical framework for supporting programmers. In: Proceedings of the European Software Engineering Conference/ACM SIGSOFT

International Symposium on Foundations of Software Engineering, pp. 351–360 (2007). DOI 10.1145/1287624.1287674
56. Ye, Y., Yamamoto, Y., Nakakoji, K., Nishinaka, Y., Asada, M.: Searching the library and asking the peers: Learning to use Java APIs on demand. In: Proceedings of the International Symposium on Principles and Practice of Programming in Java, pp. 41–50 (2007). DOI 10.1145/1294325.1294332
57. Ying, A.T.T., Robillard, M.P.: The influence of the task on programmer behaviour. In: Proceedings of the IEEE International Conference on Program Comprehenension, pp. 31–40 (2011). DOI 10.1109/ICPC.2011.35
58. Zimmermann, T., Weißgerber, P.: Preprocessing CVS data for fine-grained analysis. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 2–6 (2004)