Chapter 2, in Finding Source Code on the Web for Remix and Reuse, S.E. Sim and R.E. Gallardo-Valencia (eds.), Springer Science+Business Media New York, pp.17-34, June, 2013.

Unweaving Code Search toward Remixing-Centered Programming Support

Kumiyo Nakakoji, Yasuhiro Yamamoto and Yoshiyuki Nishinaka

Abstract Recognizing that programming is basically remixing, this chapter looks into the cognitive, social, and practical aspects of searching for and using existing code in a programming task. A code search mechanism undoubtedly plays an essential supporting role in a developers search for code in his or her own programming task. Supporting code search activities, however, demands more than code search mechanisms. At the same time, code search mechanisms also help a developer in a wider spectrum of programming activities. We present the anatomy of the cognitive activity in which a developer searches for existing code, and we propose *efficacy* and *attitude* as two dimensions depicting code search activity. We discuss areas of necessary technical and socio-technical support for code search activities in addition to code search mechanisms. We conclude the chapter by calling for a developer-centered remixing-oriented development environment.

1 Unweaving Code Search

Everything is a remix.[1] A large part of software is built by using existing code from open source software (OSS) via the Web. Programming is now viewed as basically remixing.

Kumiyo Nakakoji
Software Research Associates Inc., 2-32-8 Minami-Ikebukuro, Toshima, Tokyo, 171-8513, Japan, e-mail: kumiyo@sra.co.jp

Yasuhiro Yamamoto
Tokyo Institute of Technology, 4259 Nagatsuta, Midori, Yokohama, 226-8503, Japan, e-mail: yxy@acm.org

Yoshiyuki Nishinaka
Software Research Associates Inc., 2-32-8 Minami-Ikebukuro, Toshima, Tokyo, 171-8513, Japan, e-mail: nisinaka@sra.co.jp

[1] http://www.everythingisaremix.info/

As Henning [19] noted, development style has changed from the seventies and eighties, when developers pretty much wrote everything from scratch. A number of studies have found that developers constantly engage in searching for code, for documents, and for discussion forums. Developers almost always start their programming by searching the Web [22]. They begin to compose their own code only after making sure that there are no Application Program Interfaces (APIs) or libraries that are usable for their current tasks. Developers also use Web search results for a variety of problem-solving activities [5], and for different reasons [15], software developers use a bricolage of resources [5].

Text search mechanisms have been serving developers as essential tools. The UNIX *grep* command has been one of the most frequently used tools by developers for decades, and *emacs* has offered incremental search and another variety of text search mechanisms from the very beginning. All popular Integrated Development Environments (IDEs) are equipped with powerful text search engines.

This chapter looks into the cognitive, social, and practical aspects of searching for and using existing code in a programming task. The approach we have taken is distinguishing the issues and challenges in code search activities from those in code search mechanisms (Fig. 1).

The underlying motivation for this approach is our concern that casually claiming research on code search for software development might blur the essential aspects of a research question addressing a code search activity versus a code search mechanism. A code search mechanism undoubtedly plays an important role in supporting a developer searching for code in his or her own programming task. Supporting code search activities, however, demands more than code search mechanisms. At the same time, code search mechanisms also help a developer in a wider spectrum of programming activities, not just searching for code.

The next section presents the anatomy of the cognitive activity in which a developer uses existing code (i.e., code reuse). We propose efficacy and attitude as two dimensions for depicting a code search activity, and describe the scenarios of three typical types of code reuse by using these dimensions. The following two sections

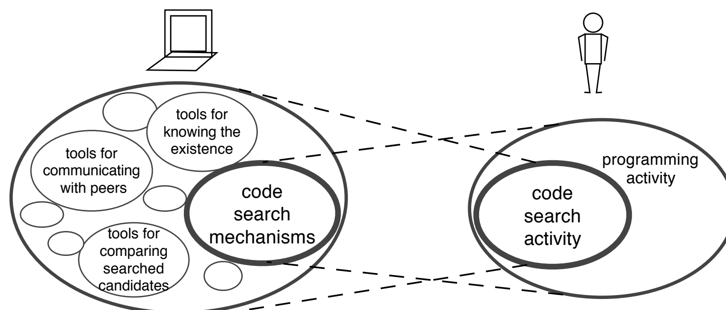


Fig. 1 Search technology is useful not only for a developer's search activities but for other development activities. A developer's search activities involve not only technology but also other computational support.

discuss areas of necessary technical support for code search activities other than code search mechanisms. Sect. 3 discusses how some aspects of code search activities call for such technical support. Sect. 4 outlines issues and challenges in using peers as information resources in code search, and addresses cognitive as well as social issues in communicating with the right person to obtain the sought information in a timely manner. Finally, Sect. 5 lists how the existing search mechanisms would further help a developer in interacting with source code.

2 Anatomy of Code Reuse

This section focuses on a developer's code search activity (see the small right oval in Fig. 1). From the cognitive perspective for the activities involved in code search, we present two dimensions for depicting code reuse[2] that underlie a wide variety of code search activities.

The first dimension is for what purpose a developer uses existing code for a current programming task. The second dimension is the way in which the developer uses existing code.

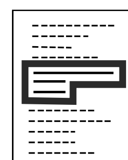
2.1 The Efficacy Dimension of Code Reuse

The first dimension, which we call the textitefficacy of code reuse, characterizes the purpose of, the motivation for, or the developer's foreseeable gain in using existing code.

There are two types of efficacy in code reuse. The first one is textitefficacy through code texture, and the other is textitefficacy through code functionality (Fig. 2).

textitEfficacy through code texture is established when a developer brings the texture (i.e., literal character strings) of an existing code segment into the program that he or she is currently editing.

[2] Note that by code reuse, we mean a developers action of simply using existing code in the developers task, and do not necessarily mean a part of more established areas of research on software reuse in software engineering.



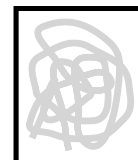


Fig. 2 The efficacy dimension: through code texture (left) and through code functionality (right)

textitEfficacy through functionality is recognized by a developer when the developer is provided with the functionality of an existing code segment that he or she needs in a program.

Different cognitive activities are involved, depending on the type of efficacy a developer seeks in reusing code. When a developer seeks efficacy through texture in reusing code, the developer must read the source code of potentially reusable parts of the code. In contrast, when a developer seeks efficacy through functionality in reusing code, the developer does not necessarily have to read its source code. Instead, the developer is likely to pay more attention to its documentation and reputation by peers and other developers.

2.2 The Attitude Dimension of Code Reuse

The second dimension, which we call the textitattitude toward code reuse, characterizes the style, or the way in which the developer uses existing code in terms of the developer's own code that he or she is currently editing.

The attitude here is along a spectrum with two ends (Fig. 3). The one end portrays the attitude of a developer when the developer uses an existing code segment textitas an element in his or her own program. The other end portrays the attitude of a developer when the developer uses existing code textitas a substrate, on top of which the developer builds his or her own program.

Needless to say, this attitude is not the property of a developer, but that of a context consisting of who is reusing which code in what task.

When a developer takes the as-an-element attitude in reusing code, he or she is likely to search for potentially usable code candidates, try using one of them, and perhaps replace it with another if the chosen one does not fit well to the task. In contrast, when exhibiting the as-an-substrate attitude in reusing code, the choice of which code to reuse would have a significant impact on the developers subsequent coding task. The developer is likely to spend a significant amount of time investigat-

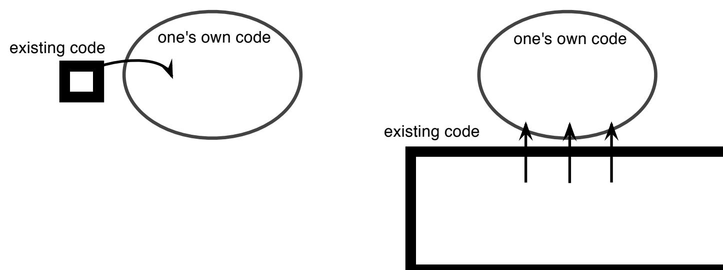


Fig. 3 The attitude dimension: along the spectrum between as-an-element (left) and as-a-substrate (right)

ing a few potentially useful candidates by carefully comparing them to decide which one to use. Once the choice is made, he or she is unlikely to replace the chosen one.

2.3 Illustrating Three Typical Code Reuse Scenarios

This subsection illustrates scenarios of three typical types of code reuse in terms of the dimensions of efficacy and attitude.

2.3.1 Searching for Previously Experienced Code

A developer often searches for a segment of code that he or she has either written before or read before to literally copy and paste it into the current code being editing (Fig. 4).

This is a typical example of a developer seeking efficacy through code texture with an as-an-element attitude.

A developer may remember previously writing a code, or reading the code (written by peer developers or by some OSS developers) that seems to be close to what he or she needs in the current program. Once the code segment is identified and pasted, the developer reads it, may modify it if necessary, or may completely discard it if it does not seem to work.

2.3.2 Using a Framework

A framework is a collection of program modules that provide a set of necessary functionality in a specific domain or service. Deciding which framework to use has a

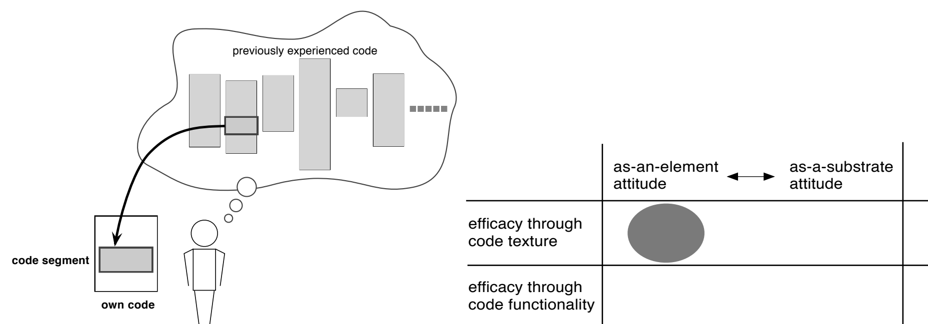


Fig. 4 Using a segment of previously experienced code. The challenge here is to find the segment of interest in vaguely remembered code locations.

significant effect on a developers programming task because it is likely to determine the architecture of the system to implement and how the program is written (Fig. 5).

This is a typical example of a developer seeking efficacy through code functionality with an as-a-substrate attitude.

Often a software architect of a software development project is the one who searches for potentially useful frameworks and decides which framework to use after comparing and studying several alternatives for the project. Once decided, it is not likely that the developer would casually replace the framework; replacing a framework possibly involves a large amount of maintenance cost.

2.3.3 Using a Code Example for a Web-Service API

A developer often uses a piece of example code for an API to use an external service (Fig. 6). This typical practice illustrates the type of code reuse in which a developer seeks both efficacy through texture (i.e., an example code) and efficacy through functionality (i.e., an external service), with an as-an-element attitude.

A developer who is interested in using a particular service functionality searches for documents and discussion forums about APIs for using that service. Typically, providers of such services promote their services to other developers and tend to provide detailed information resources for using their API, to make it easier to search for such information through Google and other Web search engines.

A rich source of examples of API usage is available on the Web [6], enabling developers to gain the efficacy of code reuse through code texture. The search result for API code examples is likely to consist of several candidates. Typically, a developer copies and pastes one of them in the current program, test-runs the program, and checks whether it successfully produces the desired effect. If it fails the developers expectation, the developer replaces the pasted part with another result from the search, and then checks whether that one works. The developer may not fully understand the pasted statement for instance, which parameter does what but if it works, he or she may just leave it as is in the program.

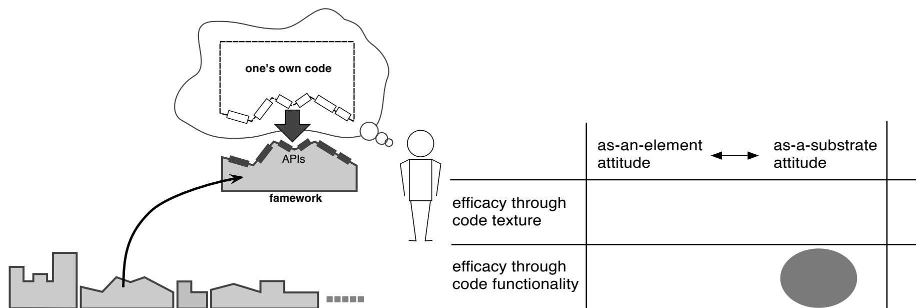


Fig. 5 Using a framework. A developer needs to develop some understanding about a framework before deciding to use it.

3 Areas of Technical Support for Code Search Activities

From the perspective of a developers cognitive activity on code search, code search mechanisms are necessary but not sufficient. Other areas of technical support should be taken into account to smoothly and effectively help a developer engaging in code reuse and remix (see the larger left oval in Fig. 1).

This section lists some aspects of a code search activity that would demand such technical support. We discuss existing tools and then outline future research agendas together with relevant existing research in addition to software engineering.

3.1 Knowing the Nonexistence of Potentially Usable Existing Code

Developers often want to make sure that there is no reusable code before they start writing their own code. Developers usually have a rather good understanding of what they want to achieve in their own programming tasks. In this sense, the code search is fact-finding rather than exploratory [25].

It remains as a challenging task to know the nonexistence of potentially usable code for the current task. When one does not find usable code among a list of queried results, it is often difficult to distinguish whether such code does not exist or whether it exists but has not been located due to inadequate queries. A developer is likely to specify a query for a desired functionality in his or her own context, but what could be suitable for the task may not be associated with such a vocabulary set [24].

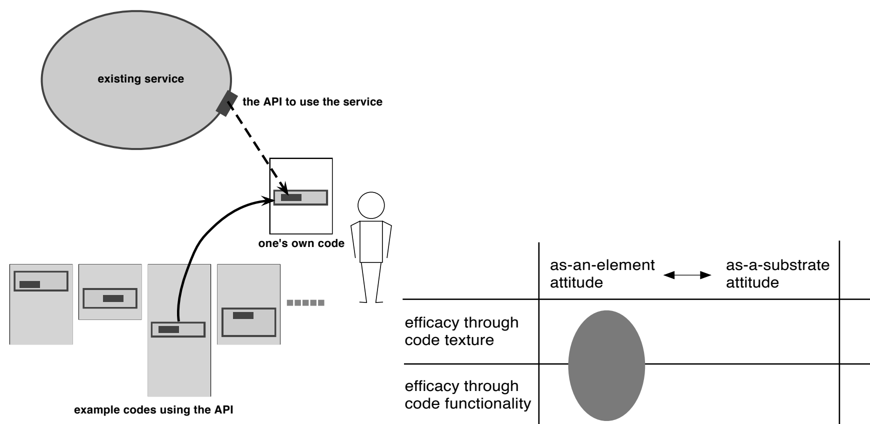


Fig. 6 Using a code example for a Web-service API. A challenge is to choose among multiple candidates the “right” example to copy that fits to the current programming needs.

3.2 Helping to Formulating Queries

Tools have been studied to address the challenge of adequately formulating queries. CodeBroker used a latent-semantic indexing mechanism to associate terms used in program comments and method signatures with methods stored in code repositories [33]. Blueprint automatically generates queries with code context for a developer and presents a code-centric view of Google search results in an IDE [6]. Mica makes it easier for a developer to explore Google search results by providing appropriate relevance cues for the information that he or she wants to know [30]. The approach by Bajracharya et al. helps developers obtain example code that uses similar set of APIs by using code usage similarity measurements, reformulating a query by showing a tag-cloud of popular words for the current query, and narrowing down results according to popularity [2]. API Explorer recommends related APIs to a developer by using API Exploration Graph, which models the structural relationships between API elements [10].

We may also resort to crowd sourcing for query formulation if we could use search query logs of tens of thousands of developers searching the Web. Such query logs are typically not publicly available but can be mined through search suggestion services [3]. Fourney et al [13] mine the Google search query log to generate a Query-Feature (QF) graph, which encodes associations between user goals articulated in queries and the specific features of a solution element relevant to achieving those goals. Using Google Suggestion and other publicly available search query suggestion services, their technique performs a standard depth-first or breadth-first tree traversal "by expanding partial queries one character at a time," starting with the keyword a user specifies. The approach has currently been applied in the CUTS (characterizing usability through search) framework, helping users to access usability information of an application system and to identify similar application systems on the Web [14]. This technique seems to be straightforwardly extendable to help developers identify potentially usable APIs, libraries, and frameworks when they are not sure about how to specify queries.

3.3 Locating a Previously Experienced Code Segment through Personal Information Management

When a developer has a particular code segment in mind to reuse, the developer may remember only its existence but not its location, or may even remember it wrongly (e.g., the code segment was written at some point of time but not saved in a file).

If a developer is unable to find the source code where he or she remembered it, the developer has to search for it. The developer may use a character string that he or she has in mind as a query, or use temporal relationships, such as that's the code I was working on when the new manager came to our project, as a retrieval cue to located potentially related files. Once presented with a searched result, the

developer immediately knows whether it is what he or she has been seeking. This type of search poses a challenge similar to the one found in the studies on personal information management [4].

Proposed tools and empirical studies in the field are applicable to help developers in this situation. For instance, using a tool such as Stuff I've Seen, which provides a unified index for email messages, Web pages, and other types of personal documents through their temporal relationships [11], a developer would be able to search the code that he or she has previously written by using his or her own experience as a contextual cue. Another example is the HCB (History-Centric Browsing)-tempo, which helps a user revisit previously visited Web pages by indexing a large volume of Web page history using the search query-based slicing (SQS) technique [29].

3.4 Using Related Documents to Judge Code Reuse Potential

A number of developers have mentioned that cognitive cost is quite high when reading source code written by someone else. A developer tends to resort to related documents and information or forums to judge whether to use a specific API, service, library, or framework, rather than reading their source code, by addressing questions such as:

- Is this API well-documented?
- How often does this library go through major updates?
- When was this answer provided in the Q&A forum?
- Which projects use this framework?
- How does this library rank in Google search? How about for the last six months?
- Which API results in more search results?
- How lively is this forum?

Discussions in forums often complement the documents created by the library/API designers by providing what the library/API owners have not tested; documents often fail to accurately explain in which contexts the library is not operational, only because it has not been tested.

3.5 Judging the Credibility of Existing Code

The number of search results on each of the few potentially usable libraries helps a developer compare which ones are more popular. Limiting the search to a certain period of time (e.g., for the period of the last six months) helps the developer to better understand the trends of their uses.

The ranking of search results helps developers infer how popular or commonly used an API is. Popularity is often a good indication of the adequate maintenance of the API/library, unless it provides a particular service or functionality.

Timestamps attached to each question and answer in discussion forums provide a valuable resource to judge whether the library is well maintained. If the information in the forum is relatively obsolete, developers may assume that an alternative library should be available that had replaced the current library, and they should start over searching.

3.6 Limiting Search Space by Using Social Information

Currently available Web search engines are good for understanding the popularity and reputation of a library and API at large, but are not helpful for a developer investigating whether it is used by those the developer trusts and respects. A socio-technical approach that combines social network systems with Web search mechanisms would significantly benefit a code-searching developer in this context.

Another socio-technical approach would be to share search histories among the members of a project, a group, an organization, or a community. The information necessary to understand a framework may have already been studied by the architect of the project when deciding to use the framework from among other alternatives. Such information that the architect had used could be valuable resources for project members when they study the framework later in the development process.

3.7 Tasting Existing Code

To examine the memory consumption or the execution speed of library APIs, developers need to try out the library in their own project context. Using tools such as Maven[3] may ease the task of building processes of a target library, but it remains cumbersome to “taste” a library API, especially if it is written in an unfamiliar programming language. Experimentability and penetrability are featured as factors affecting the usability of APIs [28], and tool support to improve such aspects is warranted.

3.8 Understanding the Implication of the Use of Existing Code before Reusing It

To decide whether to use a particular framework, developers need to have some understanding about a set of potentially complicated APIs the framework provides. APIs are the visible interfaces between developers and the abstractions provided by the framework [19]. Human-related factors, such as the design of APIs [19],

[3] http://maven.apache.org/

the usability of APIs [7], the learnability of APIs [28], and the necessary conceptual knowledge about APIs [24], are becoming key considerations in deciding which framework to use. Tools such as APATITE (Associative Perusal of APIs That Identifies Targets Easily) [12], which helps a developer browse APIs through incremental, column-based presentation of API graphs filtered by popularity of usage, have been developed to help in understanding the framework being used. However, few tools are available to help the developer understand a framework prior to starting to use it.

3.9 *Comparing the Effects of Using Texture of Existing Code Segments*

Existing program editors do not naturally support the trial-and-error process of finding a usable example code from multiple candidates. When pasted in an editor, the copied segment is merged into the existing program and becomes indistinguishable unless the developer deliberately inserts some visual cues (e.g., blank lines or comments) to designate which part is currently being tested with a pasted code segment.

Some graphic editors use a tracing paper metaphor and employ *layers* to produce variations of the current drawing area. Terry et al. explored a way to simultaneously develop alternative solutions to the current problem situation [31]. Similar techniques would help developers in the process of choosing a usable example code from multiple candidates.

3.10 *Motivation to Search Existing Code*

Although perhaps less common than in the past, a problem still remains when a developer is not motivated to search for potentially useful code. One of the early systems, CodeBroker [32], monitors a programming editor (*emacs*), infers the programmer's interest by parsing comments and message signatures, and pushes potentially relevant information to the developer in a subtle manner (i.e., in a small sub-pane located in the bottom of the *emacs* editor). The argument for such an editor is that information delivered in this way should minimize interruption, and that the system should incrementally present more in-depth information only when the developer requests it [33].

4 Socio-Technical Support for Code Search Activities

Peer developers serve as precious information resources in identifying the existence of potentially usable code through face-to-face communication [23] and in discus-

sion forums [20]. STeP_IN_Java (Socio-Technical Platform for In-situ Networking for Java programming) [34] is an example of an approach that identifies experts who might know the existence of potentially useful information for a developer. In addition, HelpMeOut suggests possibly relevant solutions made by peer developers to a novice programmer who is facing a compiler error message or a runtime exception [18].

A developer's ability to search, rank, and triage code, documents, and peer developers that are seamlessly integrated in an IDE may be ideal. However, code and documents are things, whereas peer developers are humans [34]. A challenge involves how to balance a developer's needs for communication for help in searching code with another developers' needs for a concentrated flow experience [26].

4.1 Searching for Developers for Help Is not Equal to Finding the Name of a Person to Ask

Addressing the challenge of having school teachers as knowledge resources, Illich stated that whereas a thing is available at the bidding of a user, a person becomes a knowledge resource only when he or she consents to do so, restricting time, place, and methods as he or she chooses [21].

Using humans as information resources is costly, especially if the sought humans are not dedicated to helping and providing information because they have their own time-pressing tasks.

When finding a peer developer for a question, it is necessary to search for those who not only can answer the question, but also are willing to answer the question. The search for a peer developer for an answer needs to include how to select participants for communication, what timing to use to start communication, how to invite people to participate in the communication, which communication channel to use, and how to use the resulting communicative session (i.e., communication archives) [26].

4.2 Searching for Developers Adds a Social Cost to the Cost of Information Search

Finding a person and asking him or her a question involves a social cost. Those who seek information demonstrate different asking behaviors, depending on whether they are in public, in private, communicating with a stranger, or communicating with a friend, due to the different levels of perceived psychological safety in admitting their lack of knowledge [8]. At the same time, whether developers can successfully get their questions answered depends on how they ask it, including their rhetorical strategies, linguistic complexity, and wording choice [8]. A classic study has found that engineers tend to rely on knowledge resources that are easier to access rather

than maximizing gains [16], a fact that does not seem to have changed much for the last forty years.

The perceived social burden on a potential answerer may affect how easy it is for a developer to ask a question. A field study of Answer Garden reports that because the information-seekers identity is not revealed in Answer Garden, the information-seeker feels less pressure in asking questions and bothering experts [1]. It might also become easier for an information-seeking developer to ask a question when he or she knows that the recipients have the option and freedom to ignore the request.

4.3 The Social Cost of Communication Can be Reduced by Using a Social-Aware Communication Mechanism

The benefit of a question-answer communication would primarily go to the one who asks, and the cost is primarily paid by those who are asked. Such cost includes stopping his or her own ongoing development task, collecting relevant information if necessary, composing an answer for the information-seeking developer, and then going back to the original task [36]. This type of communication is different from coordination communication, which has a symmetric or reciprocal relation in terms of cost and benefit between those who initiate communication and those who are asked to communicate [27].

A developer being asked may feel different levels of social pressure, depending on who is requesting information and through which communication channel the request is coming. For instance, it is harder to ignore a request if asked face-to-face. Developers may respond to a question not because they want to answer it but because they do not want to ignore it. Even though helping is costly, taking no action or saying “no” may also incur a social cost.

The STeP_IN framework [34] provides a social-aware communication mechanism called DynC (Dynamic Community); a temporal mailing list is created every time an information-seeking developer posts a question, with the recipients decided dynamically. Whereas the sender’s identity is shown to the recipients, the recipients’ identities are not revealed unless they reply to the request. If some of the recipients do not answer, for whatever reasons, nobody will know it; therefore, refusing to help becomes socially acceptable. If one of the recipients answers the question, his or her identity is revealed to all the members of the DynC mailing list. This asymmetrical information disclosure is meant to reinforce positive social behaviors without forcing developers into untimely communication.

4.4 Asking a Question to a Thing Relieves the Social Pressure of Communication

Social skills have been recognized as indispensable for developers, who often have to obtain necessary information from other developers using the APIs of interest both within an organization and outside of organizations over the Internet. This, however, should not necessarily be so.

STeP_IN_Java [35] allows a developer to post a question to a function API of Java libraries, relieving the developer of social pressure in posting a question to a large mailing list or identifying a peer programmer who is likely to help. The system identifies expert peer developers to the specific API by keeping track of who used the API in previous project histories, filters out the experts who have formerly had little social correspondences with the developer, formulates a temporary mailing list with the remaining experts for the question, and forwards the question to the mailing list. If someone on the mailing list posts the answer, the system brings the answer back to the asking developer.

4.5 Searching for Developers for Help Should Balance the Benefit of Individuals with the Cost of Group Productivity

Broadcasting a question may give a developer a better chance to find the right answer more quickly. However, if developers are frequently interrupted to offer help, their productivity is significantly reduced, resulting in lower group productivity.

Attention has been rapidly becoming the scarcest resource in our society [17]. We have estimated how much attention (in terms of time) is collectively spent in the Lucene mailing list and found that roughly more than 1,000 hours were collectively spent every month over the 2,282 developers [36]. In an organizational setting, this collective cost might even outweigh the benefits of developers obtaining answers from other developers. One way to reduce this cost is by limiting the recipients of the question to only those who are both able and very likely to be willing to answer the question [34].

4.6 Archiving Communication Turns the Need for Developer Search into that of Information Search

Compared to other professional communities, software developers' communities have historically taken advantage of digital media for communication, and their archives have served as rich information resources for other developers, transforming pieces of information originally available only by direct human contact into sources stored in artifacts.

Recent trends in social networking systems, such as Twitter, Facebook, and LinkedIn, open up new possibilities to collect information on a daily basis from individual developers. Such collections of information would become rich knowledge resources for developers, and make it possible to access knowledge that resides in a developer's mind without requiring social skills.

5 Interacting with Code through Code Search Mechanisms

This section briefly describes situations in which code search mechanisms help a developer interact with code (see the larger left oval in Fig. 1)

- Compose through selection
  A developer start typing the initial part of the name of an API and through IDEs that are now equipped with auto-completion mechanisms, the system incrementally searches for possible names starting with the typed string and matching the current program context. Together with a rich and fast-enough auto-completion mechanism, a developer may incrementally compose a program through a cycle of typing a few letters and selecting one of the auto-completed candidates.
- Generate clickable links
  A developer uses search mechanisms to generate clickable links to directly jump to the exact location of interest for quicker access, rather than typing a URL or navigating through browsers. In this way, a developer types the major part of the name of the library portal site and Google returns a link to the site that the developer then simply clicks for instant connection, even if the developer remembers the exact URL. In addition, a developer can insert tags in the code in Eclipse so that he or she can later search for a particular tag to which the IDE lists all the program components as clickable links to their corresponding locations.
- Translate technical messages
  A developer uses Google search as a technical translator to make sense of cryptic errors and debugging messages by searching them on the Web [5].
- Be reminded of unsure names
  A developer frequently conducts a Google search to use as a reminder of the correct name of an API or library for what the developer has in mind. Google Suggest and Google search results usually come up with a list of the variation of the name, from which a developer is likely to easily recognize the correct one.
- Debug through structural patterns
  A developer can interactively search for a specified structure in a half-billion lines of a Java program repository within one second by using CodeDepot [37], a browser-based code search tool. The developer searches for a pattern in which a specific set of APIs are called, and by looking at the searched results, he or she may notice that the specific order of the API calls is necessary.
- Observe coding standard/styles/convention
  A developer can find code segments within his or her project that do not observe a given coding standard or convention by using a structure search mechanism.

- Deal with code clones
  A developer can pay attention to the code clones of the program segment in which he or she has found a bug through CodeDepot [37] to avoid the possibility of the same bug in its cloned code.
- Be proactive to code clones
  A developer can be warned when he or she is about to edit a code segment that is likely to have clones through CloneTracker [9], a code structure search mechanism integrated within an IDE.

6 Concluding Remarks

By focusing on the remixing aspect of the coding experience, this chapter has described code search activities from a cognitive perspective. Our view is not that code is a remix but that coding is remixing. Searching for code is an essential activity in code remixing, which calls not only for search mechanisms but also for other technical and socio-technical support.

The remixing style of programming has changed through the last decade, and now developers often use existing code without knowing its details. Developers resort to documents and code examples, as well as the reputations of other developers, to judge whether to use the code for their current task. Developers take different strategies to decide when to reuse, how to reuse, and what code to reuse, depending on the efficacy they envision and the attitude they take.

Software development has come to form an ecosystem in which each developer remixes existing code to produce new systems. A developer-centered, remixing-oriented development environment must take this dynamism of programming into account. The depiction of code search activities as well as the list of technical and socio-technical needs discussed in this chapter should help in designing and building such environments.

References

1. Ackerman, M.S.: Augmenting organizational memory: A field study of Answer Garden. ACM Trans Info Sys, 16(3), 203–224 (1998)
2. Bajracharya, S.K., Ossher, J., Lopes, C.V.: Leveraging usage similarity for effective retrieval of examples in code repositories. Proc. FSE '10, ACM, New York, 157–166 (2010)
3. Bar-Yossef, Z., Gurevich, M.: Mining search engine query logs via suggestion sampling, Proc. VLDB'08, Auckland, New Zealand, 54–65 (2008)
4. Boardman, R., Sasse, M.A.: Stuff goes into the computer and doesn't come out: A cross-tool study of personal information management, Proc. CHI 04, ACM, New York, 583–590 (2004)
5. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., Klemmer, S.R.: Two studies of opportunistic programming: interleaving web foraging, learning, and writing code, Proc. CHI '09, ACM, New York, 1589–1598 (2009)

6. Brandt, J., Dontcheva, M., Weskamp, M., Klemmer, S.R.: Example-centric programming: integrating web search into the development environment, Proc. CHI '10, ACM, New York, 513–522 (2010)
7. Clarke, S.: Measuring API usability. Dr Dobbs Journal Special Windows/NET Supplement, http://drdobbs.com/cpp/184405654 (2004)
8. Cross R., Borgatti, S.P.: The ties that share: Relational characteristics that facilitate information seeking. In: Huysman M., Wulf V., Social Capital and Information Technology, pp. 137161. The MIT Press, Cambridge, MA (2004)
9. Duala-Ekoko, E., Robillard, M.P.: Clone region descriptors: Representing and tracking duplication in source code. ACM Trans. Softw. Eng. Methodol. 20, 1, Article 3, 3:1–3:31 (July 2010)
10. Duala-Ekoko, E., Robillard, M.P.: Using structure-based recommendations to facilitate discoverability in APIs, Proc. 25th European Conf. on Object-Oriented Programming, 79–104 (July 2011)
11. Dumais, S., Cutrell, E., Cadiz, J.J., Jancke, G., Sarin, R., Robbins, D.C.: Stuff I've seen: A system for personal information retrieval and re-use, Proc. ACM SIGIR Conference on Research and Development in Information Retrieval, Toronto, Canada, 72–79 (2003)
12. Eisenberg, D.S., Stylos, J., Myers, B.A.: Apatite: a new interface for exploring APIs. Proc. CHI '10, ACM, New York, 1331–1334 (2010)
13. Fourney, A., Mann, R., Terry, M.: Query-feature graphs: Bridging user vocabulary and system functionality, Proc. UIST '11, ACM, New York, 207–216 (2011)
14. Fourney, A., Mann, R., Terry, M.: Characterizing the usability of interactive applications through query log analysis, Proc. CHI 11, ACM, New York, 1817–1826 (2011)
15. Gallardo-Valencia, R.E., Sim, S.E.: What kinds of development problems can be solved by searching the web?: A field study, Proc. SUITE '11, ACM, New York, 41–44 (2011)
16. Gerstberger P.G., Allen T.J.: Criteria used by research and development engineers in the selection of an information source. J. Appl. Psych. 52(4), 272–279 (1968)
17. Goldhaber M.H.: The attention economy. First Monday 2(4), (1997).
18. Hartmann, B., MacDougall, D., Brandt, J., Klemmer, S.R.: What would other programmers do: Suggesting solutions to error messages, Proc. CHI '10, ACM, New York, 1019–1028 (2010)
19. Henning, M.: API design matters, ACM Queue 5(4), 24–36 (May 2007)
20. Hou, D., Li, L: Obstacles in using frameworks and APIs: An exploratory study of programmers' newsgroup discussions, Proc. Program Comprehension (ICPC) 2011, IEEE, 91–100 (June 2011)
21. Illich, I.: Deschooling sSociety, Harper and Row, New York (1971)
22. Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering, 33(12),971–987 (December 2006)
23. Ko, A.J., DeLine, R., Venolia, G.: Information needs in collocated software development teams. Proc. ICSE 2007, 344–353 (May 2007)
24. Ko, A. J., Riche, Y.: The role of conceptual knowledge in API usability. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Pittsburgh, PA, 173–176 (2011)
25. Marchionini, G.: Exploratory search: From finding to understanding. Commun. ACM 49(4), 41–46 (April 2006)
26. Nakakoji, K., Ye, Y., Yamamoto, Y.: Supporting expertise communication in developer-centered collaborative software development environments. In: Finkelstein, A., Grundy, J., van den Hoek, A., Mistrik, I., Whitehead, J. (eds.), Collaborative Software Engineering, chap. 11, pp. 152169. Springer-Verlag, (May 2010)
27. Nakakoji, K., Ye, Y., Yamamoto, Y.: Comparison of coordination communication and expertise communication in software development: Motives, characteristics, and needs, In: Nakakoji, K., Murakami, Y., McCready, E. (eds.) New Frontiers in Artificial Intelligence, pp. 147155. Springer-Verlag, LNAI6284 (August 2010)

28. Robillard, M.P., DeLine, R.: A field study of API learning obstacles. Empirical Software Engineering, 16(6), 703–732 (2011)
29. Shirai, Y., Yamamoto, Y., Nakakoji, K.: A history-centric approach for enhancing Web browsing experiences, Extended Abstracts of CHI2006, 1319–1324 (April 2006)
30. Stylos, J., Myers, B.A.: Mica: A Web-search tool for finding API components and examples. Proc. VLHCC '06, IEEE Computer Society, Washington, DC, 195–202 (2006)
31. Terry, M., Mynatt, E.D., Nakakoji, K., Yamamoto, Y.: Variation in element and action: Supporting simultaneous development of alternative solutions, Proc. CHI2004, ACM, New York, 711–718 (2004)
32. Ye, Y., Fischer G.: Supporting reuse by delivering task-relevant and personalized information. Proc. ICSE '02, ACM, New York, 513–523 (2002)
33. Ye, Y., Fischer G.: Reuse-conducive development environments. Int. J. Automat. Softw. Eng. 12(2), 199–235 (2005)
34. Ye, Y., Yamamoto, Y., Nakakoji, K.: A socio-technical framework for supporting programmers, Proc. ESEC/FSE 2007, ACM Press, Dubrovnik, Croatia, 351–360 (September 2007)
35. Ye, Y., Yamamoto, Y., Nakakoji, K., Nishinaka, Y., Asada, M.: Searching the library and asking the peers: Learning to use Java APIs on demand, Proc. PPPJ2007, ACM Press, Lisbon, Portugal, 41–50 (September 2007)
36. Ye, Y., Nakakoji, K., Yamamoto, Y.: Understanding and improving collective attention economy for expertise sharing, Proc. CAiSE 2008, Montpellier, France, 167–181, Lecture Notes in Computer Science, Vol. 5074, Springer, Berlin, Heidelberg (June 2008)
37. Ye, Y., Nakakoji, K.: CodeDepot: A one-stop code search environment based on the character and structural search methods, J. Digital Practices, Information Processing Society of Japan, 2(2), 117–124 (in Japanese) (April 2011)