Codebook: Discovering and Exploiting Relationships in Software Repositories

Andrew Begel
Microsoft Research
Redmond, WA, USA
andrew.begel@microsoft.com

Khoo Yit Phang
University of Maryland
College Park, MD, USA
khooyp@cs.umd.edu

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

ABSTRACT

Large-scale software engineering requires communication and collaboration to successfully build and ship products. We conducted a survey with Microsoft engineers on inter-team coordination and found that the most impactful problems concerned finding and keeping track of other engineers. Since engineers are connected by their shared work, a tool that discovers connections in their work-related repositories can help.

Here we describe the Codebook framework for mining software repositories. It is flexible enough to address all of the problems identified by our survey with a single data structure (graph of people and artifacts) and a single algorithm (regular language reachability). Codebook handles a larger variety of problems than prior work, analyzes more kinds of work artifacts, and can be customized by and for end-users. To evaluate our framework's flexibility, we built two applications, Hoozizat and Deep Intellisense. We evaluated these applications with engineers to show effectiveness in addressing multiple inter-team coordination problems.

Categories and Subject Descriptors:
D.2.9 [*Software Engineering*]: Management—*productivity* H.5.2 [*Information Systems*]: User Interfaces—*User-centered design*

General Terms: Management, Human Factors

Keywords: Knowledge management, Social networking, Mining software repositories, Inter-team coordination, Regular expression, Regular language reachability

1. INTRODUCTION

Coordination between software teams is a persistent problem in software engineering. Teams are dependent on one another for code, APIs, features, schedules, bugs and documentation [7], and require frequent and effective communication and cooperation to accomplish their tasks [13]. Unfortunately, poor execution in these areas is often a cause of inter-team conflict. The industry's move towards distributed development and increased use of technology-mediated communication only exacerbates the problems [15].

We conducted a survey at Microsoft to learn about coordination problems between software development teams. We asked survey respondents—which included software developers, testers, program managers—to prioritize 31 different information needs around inter-team coordination. The results, which we present in this paper (Section 2), show that engineers want new solutions for finding people, discovering and tracking dependencies, learning about the status of work items, and learning the rationale behind changes. What is most interesting about this list is that the **majority of indicated needs are about discovering, meeting, and keeping track of *people*, not just code.**

Software engineers are connected to one another in many ways, directly through face-to-face interactions and communication technologies, and *indirectly* through their shared work artifacts, which are stored and maintained in software repositories. Tools that discover connections inside these repositories can help address many of the engineers' coordination needs. This is an approach used by many applications in the field of mining software repositories (MSR) [14]. MSR applications typically address one particular information need at a time, for example, assigning developers to bugs [3], detecting duplicate bugs [27], or recommending related changes [32]; for more applications, see Kagdi et al.'s survey [19]. Most applications read data from only one or two software repositories and are built on very different infrastructures. These characteristics make it to difficult to address multiple information needs with a single tool or framework.

Previously, in our ICSE NIER paper, we proposed the idea of Codebook as a social networking web service that helps engineers to find and maintain connections with colleagues [6]. In this paper, we contribute the design, implementation, and evaluation of the Codebook framework. On top of this, we have built several applications which address the information needs identified by our survey.

Codebook discovers transitive relationships between people, code, bugs, test cases, specifications, and other related artifacts by mining any kind of software repository (Section 3). It extensively supports multiple information needs with *one data structure (a directed graph)* and *one algorithm (regular language reachability)*. Codebook is designed to be *customizable* by local domain experts, who have the most accurate knowledge about their teams' information needs and software development practices. These experts codify their knowledge into regular expressions that describe paths through the nodes and edges in the graph. Codebook takes care of the processing and optimization that is needed for efficient crawling, analyzing and querying of the data, even for information that is indirectly linked across repositories, a task which is inadequately addressed in prior work.

The following examples illustrate how a domain expert can write paths to help teammates discover who works most closely together:

ICSE '10, May 2-8 2010, Cape Town, South Africa

1. **Which developer owns some piece of code?**
   Regular expression: *Person Commits Changeset Modifies FileRevision Modifies Code*

2. **Which program manager wrote the specification for that code?**
   Regular expression: *Code MentionedBy WordDocument AuthoredBy Person*

3. **Which program managers and developers on the team work together (combines 1 and 2)?**
   Regular expression: *Person Commits Changeset Modifies FileRevision Modifies Code MentionedBy WordDocument AuthoredBy Person*

The results of computing these paths are pairs of people, code, bugs, test cases, specs, etc., and are revealed to front-end applications via web services. To evaluate the flexibility of our framework to address the multiple information needs identified by our survey, we have built two applications on top of Codebook. Both were designed and evaluated in consultation with the Microsoft software engineers the tools were created to help.

Our first application is Hoozizat (Section 4.1), which addresses four of the top ten information needs identified in our survey. Hoozizat is a web-based search portal that helps engineers find their counterparts who are responsible for a particular feature, API, product or service. Given some search terms, Hoozizat returns a set of related people, work items, code and files from the repositories. Next to each result is a second, shorter list of the engineers Codebook has found to be associates for people in the results, or owners for other items.

The second application is Deep Intellisense (Section 4.2), which addresses another information need from the top ten list. Deep Intellisense was first built [16] on a prior implementation [31] of Codebook. It is a Visual Studio add-in that shows a complete history of events for any program identifier that the user clicks on in the editor, including code changes, filed bugs, and forum discussions developers had about the code in question.

In addition, we specify two more applications addressing three more of the top ten information needs (Section 5). Of the remaining two needs, one is addressed in previous work [25], and one we keep for our future work.

1.1 Contributions

This paper makes the following contributions:

- The results of a *survey of inter-team coordination needs* for a variety of software team roles. (Section 2)

- A *novel, flexible, customizable framework for mining software repositories*, which can support multiple applications with single data structure and algorithm. (Section 3)

- We *demonstrated the flexibility of our framework* by building two applications which address five of the top ten information needs reported by our survey. (Section 4)

  In addition, we specify two additional applications, to address three more of the top ten needs. (Section 5)

- Microsoft engineers *evaluated the usefulness of our applications* in satisfying inter-team coordination information needs. (Section 4)

2. INTER-TEAM COORDINATION SURVEY

We conducted an anonymous web-based survey in order to learn how software engineers prioritize 31 different information needs about inter-team coordination. In June 2009 inside Microsoft Corporation, we sent an email invitation to 1,000 developers, testers, and program managers (consisting of a 3% random sample of employees in each job role). Respondents were offered a chance to win a single $250 gift certificate as incentive for completing the survey. 11% of the invitees responded to the survey.

The survey was divided into a demographic section and a section that asked respondents to check any number of 31 inter-team coordination information needs derived from previous studies [23, 21, 6, 16] and interviews with software engineers (Section 4.1). These needs were organized into eight categories: change notification, finding dependents, finding other people, finding artifacts, awareness of other teams, artifact history, work planning, and social networking. Respondents were asked to "pick the tasks that are most important to you, and where if you had a new tool that could make this task easier, it would have a big positive impact on your work day." Respondents chose an average of 12.5 tasks (SD = 5.5).

The ten most indicated coordination information needs amongst Microsoft software engineers are listed below, along with the percentage of respondents who indicated that response and a reference to where they are addressed in this paper.

1. Given a feature, API, product or service, finding out who the most relevant engineers (developers, testers, program managers, operations, leads, etc.) are in order to contact them. (83%) → *Hoozizat (Section 4.1)*
2. Finding an expert to talk to who knows a lot about a feature, API, product or service. (67%) → *Related work [25]*
3. Given a feature, API, product or service from another team, getting a list of servers, directories and repositories where the related code, bug reports, work items, specifications, etc. are located. (64%) → *Future work*
4. Finding out why a recent change was made, e.g., the related bug report/work item, specifications, or conversation threads in discussion lists. (62%) → *Deep Intellisense (Section 4.2)*
5. Being notified that there is a recent change that affects my code or work items. (60%) → *"Anxious for Awareness" (Section 5.2)*
6. Finding out who might be affected by a change I make to my code/API. (57%) → *"Who is using our code?" (Section 5.1)*
7. Finding out who owns some code or has ever worked on it in the past. (56%) → *Hoozizat (Section 4.1)*
8. Finding out who owns a specification or knows the most about it. (56%) → *Hoozizat (Section 4.1)*
9. Finding out which teams own the feature, product or service I or my team depend on. (53%) → *Hoozizat (Section 4.1)*
10. Finding out everyone outside my team who depends on my feature, API, product or service. (50%) → *"Who is using our code?" (Section 5.1)*

Notice that the top two needs are about finding the people responsible or knowledgeable in a feature, API, product or service. Five more of the top ten needs are also about finding people (5–10)! This may be to report a bug, to get programming advice, to learn about a scheduling change, to request a new feature, or perhaps to ask for a code review. Through interviews with thirteen engineers having various job roles at Microsoft, we found that most

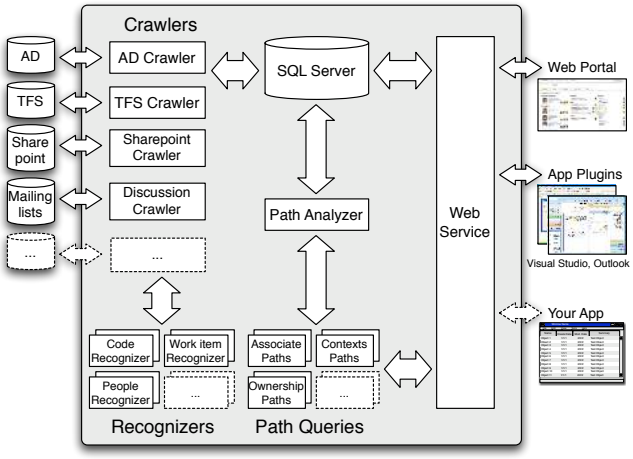

Figure 1: The architecture of the Codebook framework. Any number of repositories may be crawled, analyzed and stored as a graph in a SQL Server database. A set of paths created by domain experts is compiled, uploaded into the system, and used to discover the paths that exist in the graph. Front end applications then use web services to query the database for relevant people, artifacts, and paths that answer its end users' information needs.

ask their friends or colleagues to direct them to the people they are looking for. If their friends do not know the answer, they usually know someone else who may. This process of asking friends often resembles a game of six degrees of separation.

The people they seek can be found in the electronic repositories used by software engineers in their daily work practice. Much of that information, however, is hidden inside of these repositories and difficult to discover behind opaque query interfaces. Even after information is dug up, correlating it with information from other repositories is quite difficult [16].

Our goal for Codebook is to build a framework general enough to answer inter-team coordination information needs directly, without requiring users to conduct in-depth investigations of raw data sources on their own. In the next section, we describe the Codebook framework and discuss our solution for binding raw information together to discover and exploit the connections found within.

3. THE CODEBOOK ARCHITECTURE

The key data structure behind Codebook is a graph of typed nodes, which represent repository objects (such as people, changesets, work items, files, and source code), and typed edges, which label the relationships of the nodes to one another (such as commits, bug assignments, caller/callee, use/def, textual allusions).[1] Codebook's algorithm then walks the graph from one set of interesting objects to another via the relationship edges and discover which objects are ultimately transitively connected to each other.

To ensure scalability, it is important to not compute transitive closure blindly (which would take O(n^3) time and produce many many useless results), but instead focus on *useful paths* between pairs of nodes in the graph. A path through the graph recognized by the regular expression is *useful* if, by computing the path and its endpoint nodes, it answers a question posed by a domain expert. Paths are defined by regular expressions whose alphabet is composed of the node and edge labels in the graph. We achieve scalability by using an $O(n^2)$ *all-pairs regular language reachability* algorithm combined with an optimization specific to software engineering that drastically reduces the number of nodes considered by the algorithm.

Several steps are involved in creating a Codebook graph useful for answering end-users' questions about their team's software development activities. First, as illustrated in Figure 1, a set of crawlers mine objects from various software repositories (for example, a revision control repository, a work item database and an employee directory) and store them in the database as nodes in the graph. Relationships between these objects are derived from structure, metadata, or textual allusions and stored in the database as edges in the graph. Paths defined by regular expressions are written by domain experts and compiled by Codebook into state machines and uploaded into the database. The Path Analyzer runs a regular language reachability algorithm on the database to compute and store the start and end nodes for each path recognized by the regular expressions. To support search applications, a full text index is created for each object in the graph and its surrounding context objects (as defined by another set of regular expression paths). Finally, Codebook's data is exposed to applications via web services, enabling many different front ends to answer end-user questions with facts from Codebook.

In the next few sections, we elaborate on these steps to explain architecture and design rationale for the various parts of Codebook.

3.1 Crawlers

Codebook is designed to be an extensible system, consisting of a family of crawlers for different kinds of repositories and various types of data. As part of the prototypes we have built, Codebook can index source code repositories (Microsoft Visual Studio Team Foundation Server (TFS) and a Microsoft internal source code repository server), work item databases (TFS and a Microsoft internal work item server), employee directories (Active Directory), emails from public mailing lists (Outlook and Exchange), source code assemblies (using .NET Reflection on DLL assemblies), and web sites (Sharepoint and discussion forums).

All nodes and edges, which represent objects and the relationships between them, are stored in the database with a start date, end date, last modified date, and are uniquely named by their URI. Each object additionally contains a bag of words used for the search index, consisting of a concatenation of several strings of metadata specific to each object type. Relationships are stored unidirectionally, but paths may be defined to traverse edges in the forward or backward direction.

3.1.1 Source Code Repository Crawler

The source code repository crawlers start at the first checkin and proceed until the most recent checkin. For each checkin, the list of changed files is enumerated, and each file's differences are analyzed. Before and after snapshots of the edited files are parsed with a code analysis and compared. Whenever the differences overlap a source code element, that element is considered to have been changed by the checkin. We do not currently track code that is renamed or moved between files.

Our Codebook prototype can analyze C, C++, C#, and VBScript source code. All of the symbols (i.e., identifiers) contained within, including inside method bodies and field initializers, are stored as nodes in a database table, with metadata columns for symbol name, fully qualified name, kind (e.g., class, field, method, operator, etc.), programming language, and nesting depth. A distinct bag of words is also stored for the bodies of the definitions of symbols, such as

[1]This graph model was initially proposed by Venolia in a workshop paper [31]. Codebook is built on a direct descendent of that model.

class definitions and method definitions, to enable scoped searches.

Simple relationships between source code symbols, such as "LexicallyEnclosed" (lexical enclosure), "Superclass" (superclass and subclass links), "Calls" (method calls), "Assigns" (variable assignment), "Names" (labels), "Parameter" (parameter of a method or generic type), and "References" (appears in an expression) are stored as edges in the graph.

Relationships requiring name resolution are not directly connected. Without perfect build- or run-time information and fully linked DLLs, it is not possible to uniquely link callers and callees, or uses to their definitions. Instead, Codebook creates an intermediate non-qualified SourceCode Identifier node, and connects it to the incoming and outgoing links. This gives the added benefit that when a new definition of a method is found, Codebook does not have to add edges from all the callers of the methods with the same name to the new definition; Codebook merely connects the new definition to the already existing SourceCode Identifier node.

An additional crawler cracks open .NET assemblies and uses the .NET reflection API to read out all of the source code symbols in each DLL. The advantages of reading DLLs is that within-DLL linking is already resolved, making it possible to resolve some caller/callee and def/use relationships more precisely than when reading source code alone.

3.1.2 Employee Directory Crawler

A company's employee directory is crawled as people's names or email addresses are found in other crawlers. Each person is looked up in the directory and their name, email, title, role, department, office address, phone number, picture, and manager are stored in the Codebook database. Each person's manager is looked up as well, to create a subgraph of "Manages" relationships, all the way to the root of the management hierarchy.

3.1.3 Work Item Crawler

The work item database crawler begins at the first work item and proceeds to the most recently created work item. Since each work item may have been revised multiple times, each revision is processed separately. All work items in TFS consist of a title, a description, and a set of people who have "changed" it. The rest of the work item consists of a property bag of fields and values which should be stored in the metadata for a work item. These fields are defined by a process template *custom* to the organization which deployed the TFS repository, meaning that work item crawling must be configured by domain experts in order to understand what the fields mean. Even discovering who a bug is assigned to requires understanding the process template definition. In the repository we studied, this is in the field labeled "System.AssignedTo" whose value can be any string, not just a person.

Our prototype repository uses the Microsoft Process Template (shipped with TFS) which supports six types of work items (Value Proposition, Feature Group, Feature, Deliverable, Task, and Bug), each of which defining its own custom fields. Despite the presence of a template, a team using it can put any data they want into the fields (subject to very loose constraints). This requires that Codebook be further customized by each individual team. For example, the process template suggests the use of "System.AreaPath" to specify the component of a work item, but our team uses the field to specify the milestone for which this work item is active. They use a custom field, "Custom.01", to indicate the work item's component.

To analyze relationships in work items, Codebook must determine which fields have people, code, other work items, URLs, test cases, or a pointer to any other object inside. Codebook lets domain experts define a configuration file to specify which fields of the process template should be analyzed, and what data types each is likely to have for their team. For each field where the type is known, for example, a person field like "System.ClosedBy", Codebook looks in the employee database repository for a person matching that name or email address and creates a relationship edge between the work item and that person with the "ClosedBy" label. For fields where the type is unknown, or those which may hold natural language (like the title or description field), a set of regular expressions for each object type is run over the text. If a word is found that looks like a source code identifier (e.g. "AnnotateStringWithImage"), a "Mentions" edge is created in the Codebook graph between the work item and a non-qualified SourceCode Identifier node. We do not connect the work item directly to the SourceCode because it may not yet have been discovered by the Source Code Crawler. When new Source Code nodes are created, they are connected, if possible, to the appropriate preexisting Source Code Identifier node using a "Names" edge.

3.1.4 Textual Allusions

At Microsoft, a person's email address is often used to name the person in natural language documents, such as emails or bug reports. Whenever any short up-to-8 character word is seen in a document, Codebook looks for the word as an email address in the employee directory. If found, Codebook links that person to the object where the word was found with a "Mentions" relationship.

Textual allusions like this sometimes results in false positives. For example, an employee named "William Jones" may have the email address "will", which unfortunately, is a common English word that shows up in many emails and bug reports, not just in those that refer to William Jones. To address these overzealous connections, each Codebook graph edge has a Confidence field (a floating point number ranging from 0.0 to 1.0), that indicates the likely accuracy of the edge. Structurally-defined edges (such as lexical enclosure or bug assignment) receive a 1.0 confidence score, while other edges that derive from using regular expressions or linguistic analysis to discover email addresses or source code symbols in natural language text, receive a lower score.

3.1.5 Other Crawlers

Many useful connections can be derived from public mailing lists and discussion forums, inferring both affinity groups as well as expertise. Each message is crawled in chronological order, processing the sender and receivers of the message, as well as running regular expressions over the text to find textual allusions to other objects.

Web sites, such as Sharepoint repositories, can be crawled to find documents relevant to software development. For example, many teams at Microsoft store their specifications, meeting notes, marketing information, and legal documents in Sharepoint. The titles and contents of these documents are mined and stored in the Codebook graph and linked to the authors (who, at Microsoft, is likely to be a program manager in charge of that feature). In addition, specification documents are often constructed from templates which indicate which developer and tester will be working on the feature. Codebook's text recognizers can be customized to read that section of the document to identify the owners of the feature. The rest of the document usually contains the names of classes, methods and fields, which can be connected to the source code that eventually is written to implement the feature.

3.2 Graph Paths

A graph of related objects is the central component of many applications. For example, Facebook is centered around a graph of people who are declared to be "friends" with one another. Face-

book's graph is simple; each node is a person, and each edge is labeled "friend". Codebook's graph is more complex — there are 9 node types and 18 edges types, for a total of 29 possible triples (most are shown in Figure 2).

Due to the complexity of the underlying data that Codebook is representing, two objects may be related to one another even if they are not directly connected in the graph. For example, to find the program manager responsible for the Square method in Figure 2 (follow the **bold** nodes and edges), one needs to look for any specification documents that contain the signature of the Square method, and find its author, which in this case turns out to be Patty the Program Manager. We can further learn that Pam works with Dave the Developer because the Square identifier that Pam's specification points to was named by a method checked in by Dave. In addition, Pam created Bug #673 which is assigned to Dave. Bug #673 also includes a stack trace mentioning the implementation of the Square method written by Dave.

Though these domain-specific connections hop across many edges in the graph, they can be described succinctly by regular expressions over the node and edge labels between the paths' endpoints. *One of the key contributions of our work is to recognize that many applications previously implemented in one-off data mining software can be represented by regular expressions in this graph.*

The paths above can be written as regular expressions, starting with *"Person Authors SpecificationDocument Mentions SourceCodeIdentifier NamedBy SourceCodeMethod."* To connect that method to Dave, we add *"SourceCodeMethod ModifiedBy FileRevision ModifiedBy Changeset CommittedBy Person."* Another way to connect Pam to Dave is via Bug #673: *"Person Created WorkItem AssignedTo Person."*

The alphabet of our regular expressions are the node and edge labels from the graph. Sequences of these labels can include optional elements (`?`), loops (`+`, `*`), alternation (`|`) and grouping (`(...)`). After each edge label, the author can write a label-specific suffix (e.g. Modified*By*, or Contained*Within*) token to indicate the direction of the relationship in the regular expression. For example, in the regular expression *"Person ManagedBy Person"*, the person on the right is the manager of the person on the left.

Domain experts can both read and write these regular expressions based on their knowledge of the software development team's work practices and procedures. The paths of activity in a team where engineers work closely together in feature crews (a trio of a developer, tester and program manager) will look different than an Agile team that has no distinction between developer and tester and whose developers often pair up with new partners each day. In addition, since groups that employ the same process templates in their work item databases do not utilize the fields of these templates in the same way, having knowledge of a particular team's practices is crucial to understanding how their work is represented electronically [4].

3.3 Regular Language Reachability

Once regular expressions have been defined, Codebook computes the set of paths in the graph that conform to the regular expression. We use a modification of breadth-first search constrained by the regular expressions, an algorithm known as regular language reachability. This algorithm runs in $O((|V| + |E|)|S|)$ time for a single origin, and in time $O(|V|(|V| + |E|)|S|)$ for all origins. Codebook graphs have a power-law edge distribution — a few nodes have many many edges and the rest have few, with a long tail [31]. In the all of the graphs we have seen, $|E|$ is within two to three times $|V|$, thus we could surmise that the time complexity is $O(|V|^2|S|)$ for all-pairs regular language reachability.

Computing paths using regular language reachability provides only the endpoints of the accepted paths. Thus, one can answer the question "is there any path between A and B?", but not enumerate those paths (of which there may be an infinite number due to using loops in a regular expression). Adding in more discriminatory power to report any single path between two nodes requires a more complex algorithm, such as all-pairs shortest path constrained by a regular language, but this raises the complexity of the algorithm to $O(|V|^3|S|)$ which is impractical for large graphs of the sort Codebook creates. To compensate for our algorithm's inability to return an exact set of paths, we have found it is useful to create many short regular expressions with descriptive names. For example, the regular expression that connects a bug to a piece of code may be called "BugToCodeViaStackTraceInReport." This is almost always good enough for an end-user to believe the connection is real, and if desired, discover the exact path through inspection, now that he knows it exists.

Our implementation of regular language reachability has been executed on graphs of up to 100,000 nodes and 100,000 edges. A 53-state state machine takes about 50 minutes to compute on a dual core Intel Xeon E5450 virtual machine with 2 GB RAM, and 1 GB available to SQL Server 2008 SP1, running in Windows Server 2008 SP2 with Hyper-V.

3.3.1 Optimization

The graph described above actually contains 200,000 nodes and 350,000 edges, but we have optimized much of it away because it does not contribute to any "useful" paths. The crux of the optimization lays in pruning the SourceCode Identifier nodes.

For every source code symbol definition, the source code crawler creates two nodes, a Definition node, and an Identifier node. The Identifier node is used to connect caller/ callee, def/use chains, and "Mentions" relationships between text and code, in the face of imperfect name resolution (especially for code found in text fields).

There are two cases where the Identifier node (and its adjacent edges) are not useful for path computations. An Identifier node can exist without a definition, if a textual allusion to that identifier was made in a source code comment, work item or email, but the identifier was never realized in source code. In this case the Identifier was a mistake, and should be pruned. Second, if an Identifier node is created for a definition, but there are no "Calls," "References," or "Mentions" links to it, then the node and its edges should be pruned as well.

Performing this pruning results in a 65-75% reduction in the number of graph nodes and edges used in the algorithm. We calculated this reduction on each month of data entered into our Codebook prototype, and for each month the reduction in nodes and edges was roughly constant (+/- 5% on edges and +/- 10% on nodes). A $^3/_4$ reduction in $|V|$ and $|E|$ results in a 10-14x speedup in the running time of our algorithm.

3.4 Search

A fundamental part of the Codebook architecture is search. Abstractly, a search takes a set of keywords and returns a ranked list of Codebook nodes whose metadata best matches the keywords. We initially employed a simple search algorithm, TF-IDF (term frequency-inverse document frequency) on the node metadata's text. However, this offered poor results, since it was not possible to search for a function by the author's name, or find a person who worked with someone else who was not in his management chain.

Fortunately, the Codebook graph is much like the web in which the link structure is semantically meaningful. Thus, a better search algorithm could take advantage of this to improve the search re-

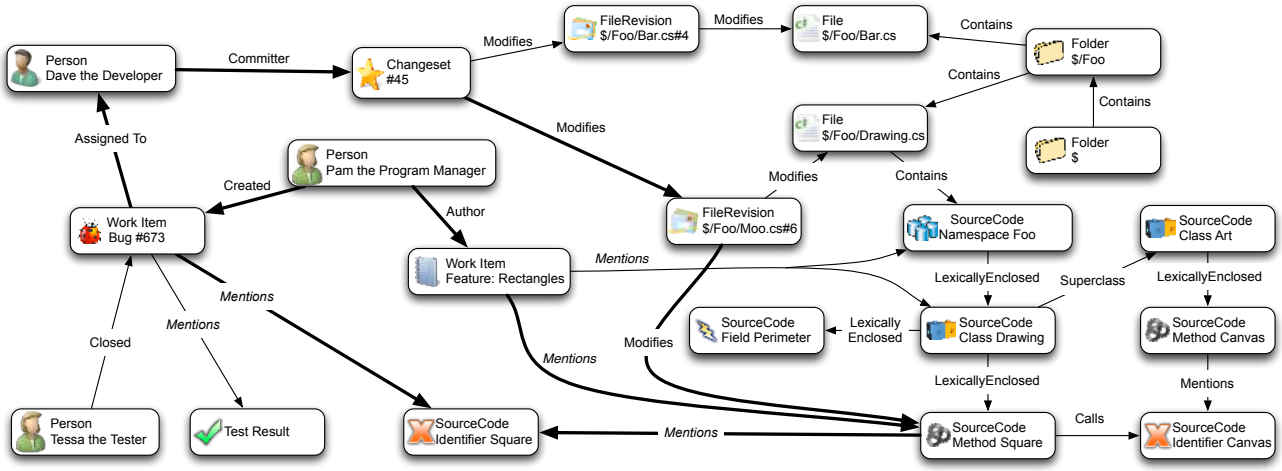

Figure 2: A canonical graph of possible relationships between objects in Codebook. "Mentions" edge labels are written in italics to indicate that they may derive from an incorrect inference. The bolded paths help explain a scenario written in the prose.

sults. Unlike the web, however, Codebook graphs do not have anchor text that describes the target of a link between two nodes, depriving us of useful context to improve search accuracy. In addition, the immediate neighbors of a node, while structurally valuable for understanding the process of the team's development tools, do not often contribute useful contextual meaning to the node.

We can use path regular expressions to find other nodes in the graph whose metadata can be used to substitute for the anchor text and immediate context we lack. Using 25 more path regular expressions, Codebook enumerates the relevant anchor nodes for each node in the graph. For example, anchor nodes include the owner of a piece of code, the person responsible for tracking a work item task, the filename where a particular source code symbol is defined, the specification document that describes a piece of code, etc. Nodes with a high degrees of "anchor" edges convey authority about a node the way that a high-degree hub does in the web.

Codebook uses SQL Server's full-text search algorithm [2] on the node and anchor meta-data to come up with a list of results. The calulcation gives us a score that we can combine with domain-specific knowledge that we learned from our interviews with developers to derive a ranking function. For example, symbol definitions are ranked higher than symbols that appear as references. Open work items are ranked higher than closed work items. People who are individual contributors rank higher than managers, since they are more likely the ones who have done the work and therefore know the most about it. Edges with lower confidence (such as connecting a bug description using the word 'will' to the Person William Jones) contribute to a lower ranking for the nodes they connect to. The specific values for these rankings can be tuned manually by the team's domain expert to conform to their software development processes.

3.5 Web Services

The final component of the Codebook architecture is a set of web services that expose the graph, the data contained within, and the computed paths to front-end clients. All nodes are referred to by URI, and their metadata can be fetched on demand. To connect a new application to the Codebook web service, the application developer uploads his path regular expressions to the system, where they are compiled and computed periodically, perhaps once per day. The developer the queries Codebook to retrieve the computed data in the form of tuples of node URIs. The developer can render his application in any form, such as a web form, a Sharepoint web part, an Outlook plugin, or a Visual Studio client add-in.

3.6 Statistics

Our Codebook repository was created from six months of development time for a medium-sized team at Microsoft. We mined their TFS source code and work item repositories, and mined employee information from the Microsoft Active Directory. The repositories we read from contained around 42 GB of data, resulting a Codebook database of 3.5 GB. It contains 420 people, 19,000 source code definitions, 9,700 files (including source code files), and 3,400 work items. The graph for this data contains 200,000 nodes and 350,000 edges and scales linearly with the age of the repository. The numbers reported throughout the paper are based on this database.

4. CURRENT APPLICATIONS

Codebook was designed to be flexible enough to satisfy a large variety of information needs using a single abstraction. In this section, we demonstrate this flexibility by describing two applications, Hoozizat and Deep Intellisense, that we built on top of Codebook.

4.1 Hoozizat: Finding People with Codebook

In the survey described in Section 2, four information needs (1, 7–9) concerned finding the people who own and are responsible for a feature, API, product or service. We built *Hoozizat*, a web search portal Codebook application, specifically to help engineers find these feature owners. Hoozizat was built in consultation with six engineers at Microsoft; we interviewed them prior to building our application to find out how inter-team coordination needs arise in their work. A typical scenario that we discovered from our interview is the following:

> Xin, the software developer, has found a bug in his code. Only Xin didn't do anything to cause the bug, other than to update a library he was using that was written by another team within his software company. He is pretty sure the bug is caused by some change to this library, but does not know whether the bug is due to his own misconceptions in using

the code, or a bug in the library itself. A typical software developer looks up the problem on an intranet or web search engine to find the answer, but in this case, since the code is not public, and the product has not yet been shipped, there is nothing to find.

Xin would like to find someone from the library team who can look at his code and tell him what is wrong with it. If it is a bug in the library's implementation, he would like to tell someone on that team to file a bug. If instead, it is a bug in the specification, he needs to find the person on the team responsible for managing the library's specification to report the problem.

Finding the right person to answer these questions involves searching through various intranet web portals such as Sharepoint, intranet search, full-text search over the codebase, and search over the company employee directory. From our interviews, we found that although one or more of these portals may point to the right answer, it is time-consuming to search multiple repositories and sift through each result list. Furthermore, to find a person that can answer Xin's questions, he would have to delve into each result to locate a related person. We found that engineers would typically spend no more than 10 minutes trying to search these web portals before giving up.

More commonly, we found that engineers such as Xin would ask their colleagues or managers if they knew a person they should talk to. When they did not, they might direct him to another person who might know. This process may repeat several times, just like a game of six degrees of separation. Xin may eventually find the right person, as long as he is motivated to put in the time and legwork. While this process is inefficient, our interviews suggest that it is much easier to find a person by asking friends than by searching through web portals.

We surmise that there are two ways we can greatly improve the search experience: first, we should search across multiple repositories at once, so that Xin would only have to query one database to find relevant answers; second, we should return not just a list of artifacts in the results, but also people that can answer Xin's questions, so Xin would not have to dig into each result to find the related person.

4.1.1 Finding the Answer with Codebook

Codebook's design is ideal for addressing this scenario. First, it crawls multiple repositories and can perform searches across all of them simultaneously. Second, Codebook can return the people Xin can talk to, not just their artifacts, by using regular language paths to describe relationships from artifacts to people. With Hoozizat, we use Codebook's built-in search, described in Section 3.4, to find a list of matching people and artifacts based on a text query.

Artifact Ownership. For artifacts returned in the search results, we augment each artifact result with a list of owners, so a user such as Xin can quickly find a person who can answer questions about that artifact. We define *owners* to be simply people who have made changes or have been assigned to an artifact:

- ***File*** *ModifiedBy FileRevision ModifiedBy Changeset CommittedBy* ***Person***
- ***SourceCode*** *ModifiedBy FileRevision ModifiedBy ChangeSet CommittedBy* ***Person***
- ***WorkItem*** *((Mentions | . . . [2] | DuplicateOf) WorkItem)∗ (AssignedTo | CreatedBy | . . . | ResolvedBy | ClosedBy)* ***Person***

[2] . . . represents additional edge labels.

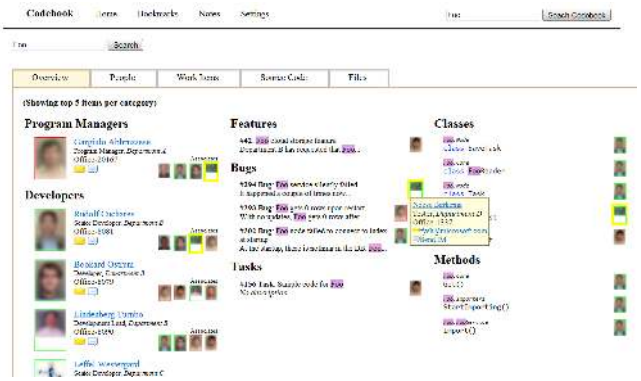

Figure 3: Screenshot of Hoozizat search results. Each column shows a different type of result, from left to right: people, work items, source code (partly hidden), and files (not shown). Small photos next to each result shows the associates for people or owners for artifacts; hovering the mouse cursor over the photos shows a tooltip with contact information for that person.

Associates. For people returned in the search results, we augment each person result with a list of associates, as this may help a user such as Xin determine the team that person belongs to, or perhaps discover another person he might know personally. We define *associates* to be people who work closely together, and from examining our Codebook repository and working with domain experts, we derived a total of 13 regular expressions for discovering them. Some relationships are simple, e.g., Alice and Bob are associates if they modified the same artifact, or more precisely, if Alice committed a changeset that modified a source code that is modified by another changeset committed by Bob:

- ***Person*** *Commits Changeset Modifies FileRevision Modifies SourceCode ModifiedBy FileRevision ModifiedBy ChangeSet CommittedBy* ***Person***

Other relationships are more complex, e.g., Alice and Bob are associates if Alice created a work item that may be a duplicate of one or more work items that mention source code that has been edited by Bob:

- ***Person*** *Created WorkItem (DuplicateOf WorkItem)∗ Mentions SourceCodeIdentifier NamedBy SourceCode ModifiedBy FileRevision ModifiedBy Changeset CommittedBy* ***Person***

Note that in the above examples, the path regular expressions are almost literal translations of their descriptions. Based on our experiences and interviews with domain experts, we believe it should be easy for a domain expert to describe such relationships.

4.1.2 Presenting Search Results with People

We present the search results in a web-based interface shown Figure 3. The first column shows people results, while the remaining columns show other artifact types: work items, source code (partly hidden), and files (not shown).

Next to each search result, we show a small list of photos corresponding to the associates for people results, and owners for other artifacts. Hovering the mouse cursor over a photo brings up a tooltip with contact information for that person including action items to send an email or an IM to that person. This allows the user to quickly identify the people most closely related to the search results and establish communication with that person. Additionally, hovering over a photo also highlights all occurrences of that person

in the search result, allowing the user to discover other people or artifacts to which that person may be related.

4.1.3 Evaluation

To evaluate the correctness of the Hoozizat application, we interviewed five engineers whose data is contained within our prototype Codebook repository. We continued our interview with these five, and an additional nine, engineers to evaluate the utility and design of the user interface.

During each interview, we explained the Codebook project and demonstrated the functionality of the Hoozizat interface using a variety of searches we had learned during our testing showed a lot of varied results. With each of the five stakeholders, we asked them to type in searches for their name, some function names from their code, and some keywords from their features and bugs. Participants each did 5-6 searches, and in each case, pointed out to us that the people in the result list were their colleagues in development or program management (none of their searches resulted in any testers being returned). One said *"All of these people work near me in the same hallway."* They indicated that all of the code they saw was part of their project and was the code they themselves wrote. Likewise, for the work items (features and bugs), they told us that they were indeed assigned to work on those features, or were in fact the owners of the work items. This shows that Codebook was able to return results that were recognizably correct to people whose data was in the system.

While the interviewees said that no results were missing from Codebook's result set, four out of the five interviewees indicated that several people and work items returned from their Codebook searches should not have been in the list. These false positives showed up due to inaccurate textual allusions between English language words and people's email addresses (e.g., William Jones' email address is "will," and he showed up in a lot of searches). We have used the incidence of these false positives to improve our search ranking function and penalize links from common English words to people or source code identifiers.

We learned more about the team's development practices from talking to the stakeholders. One program manager explained to us how we could tell from who was assigned to a feature where it was in the process of being implemented. A work item of type feature with a program manager (PM) was likely just an idea that got cut. A feature with a PM and a developer meant that it was being implemented. A feature with a PM, developer and a tester meant that the feature was complete, and in testing and stabilization.

Hoozizat's interface was well received. All 14 engineers found the ability to search across multiple repositories and artifact types to be very useful. Six asked why these particular results were chosen to be returned. They wanted to see not just that two items were related to one another, but the path that connected them. Everyone liked that Hoozizat shows the associates and owners for each result, noting that it resembles social networking applications like Facebook which are popular today. One **developer** said, *"You feel like you're alone coding in your office, but now with Codebook when something happens to the code or bugs you'll feel less bored."* A **manager** commented on cutting out the middleman in his quest for answers, *"the more you can help me get my job done without talking to [too many] people, the faster I can go."* Nine out of 14 of the engineers expressed a remarkable amount of devotion to Hoozizat at the end of their interviews. One engineer said to us, *"I don't know how to live without this."*

Each of the engineers had different opinions about how to rank the search results from each category, though generally people from the same team explained similar ranking beliefs. This reinforces our belief that tools for software engineers must be customizable by domain experts and end-users if they are to be successfully adopted.

4.2 Deep Intellisense

Deep Intellisense [16] is a Visual Studio add-in to aid code investigation, which was ranked fourth on our survey of information needs. When the user clicks on any source code symbol in the editor, it displays a reverse chronologically sorted list of events showing everything that has happened to that source code symbol in the development history, including code changes, work items and messages from discussion forums that refer to it. These are discovered by the following paths, all starting from SourceCode nodes.:

- *SourceCode ModifiedBy FileRevision ModifiedBy* ***Changeset***
- *SourceCode MentionedBy SourceCodeIdentifier MentionedBy* ***Changeset***
- *SourceCode MentionedBy SourceCodeIdentifier MentionedBy* ***WorkItem***
- *SourceCode MentionedBy SourceCodeIdentifier MentionedBy* ***DiscussionForumPost***

Deep Intellisense also displays the people associated with each artifact, including their role and contact information. Additional information about this scenario can be found in our MSR paper [6].

Like the process we undertook with Hoozizat, Deep Intellisense was designed with participation from five developers and testers at Microsoft, who were interviewed to understand their work practices and information needs around code investigation, and to give us feedback on mockups of our user interface.

Deep Intellisense was prototyped on three large projects (CKS, Rawr, and AjaxControlToolkit) from Microsoft's open-source repository, CodePlex.com, and demoed to software developers at Microsoft's Professional Developer Conference in September 2008. Feedback was universally positive, with most participants eager to see the feature deployed on their own company's software projects.

5. OTHER APPLICATIONS

In addition to our current applications, Codebook can be used to build many other applications. In this section, we describe two other Codebook applications, "Who is using our code?" and "Anxious for Awareness," and the path regular expressions required to implement them.

5.1 Who is Using Our Code?

In a company that produces both applications and frameworks, a framework team may not be aware of every other individual or team who is using their framework. This makes it difficult to notify dependents when breaking changes must be made (information needs #6 and #10 on our inter-team coordination survey). Codebook can be used to mitigate this issue by discovering everyone who may be affected by breaking changes, e.g., by discovering when a person (in the team) edited some source code which is called by code edited by another person (outside the team):

- ***Person*** *Committed ChangeSet Modifies FileRevision Modifies SourceCode CalledBy SourceCode NamedBy SourceCode ModifiedBy FileRevision ModifiedBy ChangeSet CommittedBy* ***Person***

Once these paths are computed, Codebook can easily filter the results of this regular expression to restrict the person at the beginning of the path to be people inside a team, and the person at the end at the path to be people outside that team. The user interface

would also provide action items, such as a link for contacting the owners of all calling methods in order to inform them of breaking changes.

5.2 Anxious for Awareness

When teams collaborate as part of a large project, a member of one team will often assign a work item to a member of another team. Tracking the status of work items assigned across teams is frustrating because the teams' independent work is not transparent to each other (information need #5 from our survey). The work item can be delayed due to poor communication, differing priorities, or forgotten altogether because no one advocates for it [7]. Codebook can help increase transparency between teams by discovering people who have referred to the work item from another work item, people who have worked on code mentioned by related work items, or source code changed by related work items:

- *WorkItem ((Mentions | ... | DuplicateOf) WorkItem)∗ (AssignedTo | CreatedBy | ... | ResolvedBy | ClosedBy)* ***Person***
- *WorkItem ((Mentions | ... | DuplicateOf) WorkItem)∗ Mentions (SourceCode ModifiedBy FileRevision ModifiedBy)? Changeset CommittedBy* ***Person***
- *WorkItem ((Mentions | ... | DuplicateOf) WorkItem)∗ Mentions Changeset Modifies FileRevision Modifies* ***SourceCode***

Once the work item has been assigned, one could follow a newsfeed of the assignee's activities and watch his progress on the work item. Browsing the assignee's team's newsfeed could provide context about the team's changing deadlines and priorities.

6. RELATED WORK

We first describe our own related work which motivated and led to the Codebook framework. Other related work falls into fields of Semantic Web and software engineering.

Codebook is not the first work in the software engineering field to mine software repositories and to query graphs. However, Codebook is the first work to combine *multiple repositories* within one graph to support *multiple applications*, while still allowing *powerful analyses* (multi-hop relationships), *customizability,* and *extensibility*. Other related work has satisfied some, but not all of these criteria.

6.1 Own work

The Bridge was our team's first prototype of a graph consisting of people, code, bugs and emails derived by crawling software development repositories. The Bridge exposed its data via a strongly typed API which made access to values in the graph straightforward, but incremental changes to the schema difficult. Codebook has inherited the Bridge's ability to scale to large repositories (millions of nodes and edges), but has been modified to enable easier access to the underlying data and greater customizability. End-user application methods can now be implemented directly via regular expression paths.

Deep Intellisense, described in Section 4.2, was originally built on the Bridge. During this effort, we learned that applications must be customized to the distinct needs of each development role (developer, tester, manager). In addition, scoping information to source code symbols rather than files was an important way to match the developer's tasks. Both insights have been taken into account when designing Codebook. Our experience from building Deep Intellisense was one source of inspiration to use regular language reachability as Codebook's core analysis, to make it easier to customize and build a wider variety of applications.

6.2 Semantic Web

The Semantic Web uses RDF triples [24] to describe the semantics of documents, people, or any type of object accessible by an URI. RDF triples are clauses of the form *<Subject,Verb,Object>* which form a graph of nodes (subject and object) and edges (verbs). A SQL-like query language called SPARQL [26] is used to look up nodes and edges in the RDF graph. Several extensions to SPARQL, such as PSPARQL [1] and SPARQLeR [22], have been proposed and developed to support resolving paths through an RDF graph. Codebook takes a similar approach towards the use of regular expressions (REs) to define paths through its graph. CSPARQL [1], an extension of PSPARQL, adds constraints to the REs, giving the ability to specify a type for the node and a constraint on its value. Codebook does not yet support user-customizable constraints.

Kiefer, Bernstein and Tappolet use RDF and an extension to SPARQL to discover patterns of similarity in software repositories [20]. This approach is similar in concept to our own, but does not take advantage of paths through graphs to discover relationships. With their use of an in-memory data structure, the current implementation of their approach has limited scalability, and their performance is, in their own words, "not satisfactory."

Hyland-Wood, Carrington and Kaplan [18] propose to use RDF and SPARQL as a mechanism to discover single-hop relationships in a graph derived from software maintenance information. They implemented a proof of concept for an example consisting of only two object-oriented classes. In contrast, Codebook scales to very large software projects and exploits multi-hop relationships.

6.3 Software Engineering

In their Hipikat tool [11], Cubranic and Murphy used a graph of change tasks, file versions, people, messages, and documents to recommend related software artifacts by following a single relationship in the graph. Alex Tarvo used a similar graph of bug reports and file versions in his BCT tool [29] to predict software regression. The Fran tool by Saul, Filikov, Devanbu, and Bird walks call graphs to find related functions, but only two steps [28]. In contrast to all these works, Codebook supports multi-hop queries (i.e. more than two steps) and more than one application through its use of regular language reachability. In the Codebook framework, applications can also be easily customized to the specific needs of development teams.

Grok and other Prolog-like languages [17] support pattern discovery using relational algebras defined over graphs of tuples. The graphs, which derive solely from source code analysis, have been mined for design patterns, architectural and protocol compliance, and change impact. While Codebook's REs have less power than relational algebras, we have found them adequate to describe all of our applications' information needs. In additions, REs are less complex, ensuring that pattern creation and comprehension are as accessible as possible to our end-users.

DebugAdvisor [5] is a tool that supports debugging activities by allowing free-text queries over an index of structured software repository data. First a "fat query" is analyzed and turned into a hierarchical tagged structure of bags of words. Each bag is used to query an associated repository, returning a graph of likely nodes in the index. The links in the graph are then analyzed, resulting in a single ranked list of result nodes. In short, DebugAdvisor uses a graph to combine and rank results from several different types of data. In contrast, Codebook traverses a graph to find exact matches to queries written as path regular expressions. The main focus of DebugAdvisor is debugging, while Codebook's focus is on improving inter-team coordination by supporting multiple applications with a single framework.

Many applications can be built on top of our Codebook framework. For example, the Augur tool by de Souza et al. [12] provides visualizations of developer activities during the software development lifecycles. Among other visualizations, de Souza et al. show in a "social" call graph how developers are related to one another through the code that they wrote and call. The Ariadne tool by Trainer et al. [30] displays a similar social call graph. In Codebook, a possible regular expression to compute a social call graph is *Person Modifies Code Calls Code ModifiedBy Person*.

The Expertise Browser by Mockus and Herbsleb [25] addresses information need #2 from our survey. It can also be built on top of Codebook (*Code ModifiedBy Person*). The analysis of socio-technical congruence [10, 9, 8] is supported by Codebook as well: for the technical dimension, recorded dependencies help to identify coordination needs (e.g, *Code Calls Code*); for the social dimension, coordination activities are recorded directly (e.g., *Person Modifies WorkItem ModifiedBy Person*).

7. CONCLUSION

In this paper, we address the problem of inter-team coordination with ***Codebook, a framework for connecting engineers and their work artifacts together.*** We motivated our work with a survey of software engineers at Microsoft who helped us prioritize the most important information needs around coordination that should be addressed by new tools. We designed our framework around a single data structure (a directed graph) which captures the relationships between people, code, bugs, specifications, and other work artifacts that are mined from any number of software repositories. We discover transitive connections using a single algorithm (regular language reachability), which, with our optimizations, scales to large graphs, and enables the customization of Codebook by domain experts to fulfill the information needs of their teams, on the data that their teams have recorded in the repositories. In the future, we plan to augment our regular language reachability algorithm with the ability to compute a total weight per path, and to combine these weights to make stronger inferences about the veracity of connections between people and artifacts in the graph. We built two front-end applications, Hoozizat and Deep Intellisense, to demonstrate the effectiveness of our framework, and have plans to build several more.

Using our Codebook framework, software engineers no longer have to dig through repositories or pester their colleagues to discover, track and maintain connections to other people and their associated work artifacts. It is an important step on the way to address the challenges of inter-team coordination.

8. REFERENCES

[1] F. Alkhateeb. *Querying RDF(S) with Regular Expressions*. PhD thesis, Joseph Fourier University of Grenoble, June 2008.

[2] M. C. Andrew Cencini. Sql server 2005 full-text search: Internals and enhancements. http://msdn.microsoft.com/en-us/library/ms345119(SQL.90).aspx.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of ICSE*, pages 361–370, 2006.

[4] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of ICSE*, pages 298–308, 2009.

[5] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala. Debugadvisor: A recommender system for debugging. In *Proceedings of ESEC/FSE '09*, August 2009.

[6] A. Begel and R. DeLine. Codebook: Social networking over code. In *Proceedings of ICSE, NIER Track*, 2009.

[7] A. Begel, N. Nagappan, C. Poile, and L. Layman. Coordination in large-scale software teams. In *Proceedings of CHASE*, pages 1–7, 2009.

[8] M. Cataldo, D. Damian, P. Devanbu, S. Easterbrook, J. Herbsleb, and A. Mockus. 2nd international workshop on socio-technical congruence, May 2009.

[9] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of ESEM*, pages 2–11, 2008.

[10] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of CSCW*, pages 353–362, 2006.

[11] D. Cubranic, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE TSE*, 31(6):446–465, 2005. Member-Gail C. Murphy.

[12] C. de Souza, J. Froehlich, and P. Dourish. Seeking the source: software source code as a social and technical artifact. In *Proceedings of GROUP*, pages 197–206, 2005.

[13] C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of ICSE*, pages 241–250, New York, NY, USA, 2008. ACM.

[14] A. E. Hassan. The road ahead for mining software repositories. In *Proceedings ICSM, FoSM track*, pages 48–57, 2008.

[15] P. Hinds and C. McGrath. Structures that work: social structure, work structure and coordination ease in geographically distributed teams. In *Proceedings of CSCW*, pages 343–352, 2006.

[16] R. Holmes and A. Begel. Deep intellisense: a tool for rehydrating evaporated information. In *Proceedings of MSR*, pages 23–26, 2008.

[17] R. C. Holt. Grokking software architecture. In *Proceedings of WCRE*, pages 5–14, 2008.

[18] D. Hyland-Wood, D. Carrington, and S. Kaplan. Toward a software maintenance methodology using semantic web techniques. In *Proceedings of SOFTWARE-EVOLVABILITY*, pages 23–30, 2006.

[19] H. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.

[20] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with iSPARQL and a software evolution ontology. In *Proceedings of MSR*, page 10, 2007.

[21] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of ICSE*, pages 344–353, 2007.

[22] K. Kochut and M. Janik. Sparqler: Extended sparql for semantic association discovery. In *Proceedings of ESWC*, pages 145–159, 2007.

[23] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of ICSE*, pages 492–501, 2006.

[24] F. Manola and E. Miller. RDS primer. http://www.w3.org/TR/REC-rdf-syntax/, February 2004.

[25] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of ICSE*, pages 503–512, 2002.

[26] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/, January 2008.

[27] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of ICSE*, pages 499–510, 2007.

[28] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of ESEC-FSE*, pages 15–24, 2007.

[29] A. Tarvo. Mining software history to improve software maintenance quality: A case study. *IEEE Software*, 26(1):34–40, 2009.

[30] E. Trainer, S. Quirk, C. de Souza, and D. Redmiles. Bridging the gap between technical and social dependencies with ariadne. In *Proceedings of eTX at OOPSLA*, pages 26–30, 2005.

[31] G. Venolia. Textual alusions to artifacts in software-related repositories. In *Proceedings of MSR*, pages 151–154, 2006.

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