Developer-Centric Models: Easing Access to Relevant Information

Thomas Fritz
Department of Computer Science
University of British Columbia
fritz@cs.ubc.ca

ABSTRACT

During the development of a software system, large amounts of new information, such as source code, work items and documentation, are produced continuously. As a developer works, one of her major activities is to consult small portions of this information pertinent to her work to answer the questions she has about the system and stay aware of the relevant information. Current development environments are centered around models of the artifacts used in development, rather than of the people who perform the work, making it difficult and sometimes infeasible for the developer to satisfy her information needs. To support developers in managing the information, we introduce the concept of a developer-centric model and introduce two novel developer-centric models. The Degree-of-Knowledge (DOK) model provides a means to automatically determine the core of what a developer knows. This knowledge model can then be used to identify the information a developer might be interested in. The information fragment model supports the automatic integration of information to help rank, filter and interpret the information a developer might be interested in.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

Keywords

Human-centric software engineering

1. PROBLEM AND MOTIVATION

In the development of a software system, large amounts of new information are produced continuously. Source code, bugs, iteration plans and documentation, to name just a few, are changed or newly created by developers of the software system every day. As part of producing this new information, a developer must continuously answer questions about the current state of the project [24]. Answering these questions typically requires searches to be performed over the large amounts of system information to find the small portion that is pertinent to the developer's work. One of the commonly asked questions Ko and colleagues found by observing seventeen professional developers at Microsoft is "What have my coworkers been doing?" [18]. Answering this question requires finding just the changes completed by members of one's team amongst the many changes that may have been completed by the entire software development staff for the project. Given the long history of integrated development environments (IDEs) used by software developers and the many capabilities in these environments, one would expect that answering a question specific to a developer's needs would be easy.

Unfortunately, answering such questions is not straightforward because today's development environments are centered around models of the artifacts used in development, rather than of the people who perform the work. For instance, a central model in a development environment is an abstract syntax tree, which provides an abstraction of source code to facilitate feedback about the syntax of code being written and to facilitate code transformations. By providing feedback about artifacts, these models benefit the developers using the environment. However, these models fall short for developers by supporting only a small fraction of their information needs when they are performing work on the system. In particular, instead of helping a developer pinpoint just that information needed to perform work at the moment, development environments have been designed and engineered to provide information about all artifacts associated with the system. The result for the developer is that it is often difficult, and sometimes even infeasible, to find the answer to a question of interest.

The hypothesis of this research is that *developer-centric models can be combined with artifact-centric models in a development environment to ease a developer's access to the information relevant to the work-at-hand.* Specifically, we introduce two novel developer-centric models to support a developer in answering a broader set of questions related to her information needs than is possible with existing research. One model, the *degree-of-knowledge model*, represents a developer's knowledge of the source code comprising a system of interest. This model can be used to identify who the developer can ask to find out about parts in which the developer lacks knowledge, a situation that arises frequently on projects [20], and to identify information about the system in which the developer might be interested, which is important to stay aware of changes that might break the code [19]. The second model, the *information fragments model*, provides a means for a developer to integrate and

compose information about the system to answer questions that are infeasible to answer with current approaches [18].

More details on our research and most of our results are presented in [12, 13, 11, 9, 10].

2. BACKGROUND AND RELATED WORK

For many years, to help manage the complexity of building software systems, software developers have built and relied upon the services of integrated development environments. Early development environments, such as Interlisp [1], provided developers with information about how the software they were constructing connected together, such as which functions called which other functions. Later efforts, expanded the capabilities of development environments beyond easing navigation across source code to include such concepts as configuration management (e.g. [2]).

This previous research has resulted in development environments that provide significant aid to software developers. However, these development environments are centered around models for the artifacts, making it difficult or infeasible to answer questions that address a developer's individual information needs. The two models we introduce in this thesis increase the information needs of a developer that can be satisfied within a development environment.

2.1 Degree-of-Knowledge Model

The DOK model we introduce seeks to support developers by modeling a developer's knowledge and supporting a developer in situations such as finding the expert of a part of the code or identifying the changes of interest to the developer. Researchers have proposed support for answering the question of *who to ask* through recommenders that suggest experts for parts of the code based on the authorship of changes to the code (e.g. [21, 20]). A tacit assumption with these existing recommenders is that a developer's changes to the code indicate her knowledge about the code. These approaches focus on a system-wide knowledge model for code, trying to find one expert for each code element. By focusing on the code instead of the developer, these approaches fall short by assuming that authorship is the only way that a developer can gain knowledge about code. In fact, developers must interact with the code prior to authoring the code. These approaches also treat a developer's knowledge as a monotonically increasing function whereas, in reality, a developer's knowledge ebbs and flows as different developers change the same part of the code base. Our DOK model differs from previous expertise identification approaches by modeling the ebb and flow and by considering not just the code a developer authors and changes, but also code that the developer consults during their work.

Researchers have also previously considered the question of *which change should a developer know about* by flowing all notifications on changes to developers through such mechanisms as mailing lists and commit logs [14] or by displaying the changes directly in the developer's environment [23]. These approaches place the burden on the developer to identify changes of interest. Given how much information can change in a software system within a short amount of time, this identification can be time-consuming and tedious [14]. The DOK model automatically identifies changes of interest to a developer.

Researchers have also more generally studied how to explicitly model the different parts of a developer's knowledge and how developers comprehend code (e.g. [26]). These approaches focus on the mechanisms of gaining knowledge rather than what knowledge a developer retains as a result.

2.2 Information Fragments Model

The degree-of-knowledge developer-centric model provides a developer's perspective on one kind of information in the development environment, namely source code. A developer also has information needs when working on the system that span across multiple kinds of information. With existing development environments that put different kinds of information into different silos [6], answering this type of questions is difficult.

While earlier work points at the need for answering questions across multiple kinds of information [18, 19], this work states questions at an abstract level and does not discuss the ambiguity that lies in the developer's interpretations of these questions. To better understand the range of questions of this form that are asked by developers and to better understand the concrete forms of the questions, we performed an exploratory study, as part of our research, identifying 78 questions of interest to developers and pointing out the variations in developer's interpretations.

One previous approach to help developers answer questions that span across multiple kinds of information is to provide a query language supporting queries across all the kinds of information involved (e.g. [17, 22]). In this case, the developer must know or learn the query language and must specify explicitly how the different kinds of information should be integrated. Different to this approach, the information fragment model automatically composes information of interest.

Another approach to support such questions is to provide a fixed integration of different kinds of information through such means as a fixed schema (e.g. [25]), an explicit ontology that models different kinds of information and relations between the kinds (e.g. [15]) or by overlaying the information in a view (e.g. [8]). These approaches restrict the way information can be integrated and thus limit the flexibility required to answer the multitude of questions. In contrast, our information fragments model approach focuses on allowing the user to compose and view different kinds of project information to support the many questions that arise during a work day.

3. APPROACH AND EVALUATION

We introduce two developer-centric models, the degree-of-knowledge model and the information fragments model to support developers in accessing the small portions of information needed to answer the questions they have. We now proceed to describe each of these models and their evaluation in more detail, before we shortly describe how these two models can be synergistically combined to support a developer in staying aware of relevant information.

3.1 The Degree-of-Knowledge Model

To address limitations in earlier efforts, we developed the degree-of-knowledge (DOK) model that approximates an individual developer's knowledge of the system's code. Similar to artifact-centric models, the DOK model can be embedded within a development environment and can be determined automatically.

As an initial step towards a model, we performed an exploratory study with 19 professional software developers to investigate the factors that should be used in modeling a developer's knowledge. In this study, we investigated a developer's interaction with the code as a proxy for her knowledge, since interactions always precede and are part of authoring changes. The study establishes, with statistical significance, that the more frequently and recently a developer interacts with a code element (as represented by a high degree of interest (DOI) value [16]), the more the developer knows about the element. From interviews with 13 of the 19 developers, we also determined a number of other factors that may be used to model a developer's knowledge, such as the authorship of program elements and the code stability.

Given our pursuit to define a model that can determine automatically a developer's knowledge of code, we chose to focus on the interaction and authorship aspects. We gathered data in another study, from seven professional developers that provides evidence that authorship and interaction each capture a unique and valuable perspective on a developer's knowledge. Authorship represents a longer-term component of a developer's knowledge; interaction indicates a shorter-term component.

Using the results from these two studies, we developed the degree-of-knowledge (DOK) model to capture a developer's individual knowledge of code. The DOK model assigns a real value to each code element–class, method, field– in a developer's development environment and is based on a linear combination of a developer's interaction with an element, computed by the DOI, and three factors of authorship: whether the developer was the first author of the element (FA), the number of changes the developer has contributed to the element (DL) and the number of changes others have made to the element (accepted changes – AC).

$$DOK = \alpha_{FA} * FA + \alpha_{DL} * DL + \alpha_{AC} * AC + \beta_{DOI} * DOI$$

This combination accounts for knowledge gained by a developer interacting with the code for such purposes as trying to understand how the code functions as well as the knowledge that results from creating the code. This model also accounts for the ebb and flow of a developer's knowledge; for instance, a developer's knowledge of a code element decreases when someone else changes the element. The DOK model enables the automatic computation of the individual knowledge of each developer in a source code element by combining authorship data from the source revision system and interaction data from monitoring the developer's activity in the development environment. We used a linear combination as an initial starting point.

To empirically determine initial weighting factors for each of the four factors (FA, DL, AC, DOI), we conducted an experiment, with seven developers. In essence, the experiment involves gathering data about authorship from the revision history of a project, about interest by monitoring developers' interaction with the code as they work on the project and about knowledge by asking developers to rate their level of knowledge of particular code elements. Using the developers ratings, we then applied multiple linear regression. The best fit of the data was achieved with the values presented in Table 1. We found that both authorship and interaction improve the quality of the model and help to explain a developer's knowledge of an element.

Table 1: Coefficients for Linear Regression

	Weighting	Std. Error	p-value
Intercept	3.293	0.133	< 0.001
FA	1.098	0.179	< 0.001
DL	0.164	0.053	0.002
$ln(1 + AC)$	-0.321	0.105	0.002
$ln(1 + DOI)$	0.190	0.100	0.059

3.2 DOK Model Evaluation

The availability of an individual DOK model for each developer in a team opens up several possibilities to improve a developer's productivity and quality of work. We considered three possibilities through exploratory case studies that we conducted with three different teams at three different sites. These studies consider three questions that were raised by others as important in the literature: who should I ask about a part of the system's code base [20], which changes to the code base should I know about [7] and what code do I need to know about [4].

We found that the DOK model outperforms existing approaches for finding the expert in parts of a code base across teams with different code ownership styles. We also found that the DOK model can help to accurately identify changes of which a developer should likely be aware and learned about kinds of source code, namely API elements, for which our current definition of DOK does not adequately reflect a developer's knowledge.

To examine whether the DOK model is of value in other environments, such as different teams, different project phases or different working styles, we collected data from two more teams from different sites. We found that despite their significance, differences in the authorship and interaction behavior between the teams only have a minor impact on the value of the DOK model.

Despite the simplicity of the linear regression and the model itself, the degree-of-knowledge provides value in different scenarios a developer faces. This initial evidence suggests that further study of the DOK model is warranted. In particular, the weightings for the factors contributing to the DOK model and the appropriate amounts of data to use to compute DOK values require experimentation with more developers in a greater variety of situations. Extending the model with more structural information as well as increasing the granularity of the approach, such as taking into account the number of lines being changed rather than just looking at the element level, could help to improve the accuracy of the model and better capture API elements.

3.3 The Information Fragments Model

To investigate the range of developers' questions that span multiple kinds of information, we interviewed eleven professional software developers. Using an open coding method, we identified 78 questions of interest to the developers that have the characteristic of requiring multiple kinds of information to answer together with the answers the developers desired for these questions. These questions span eight domains of information: source code, change sets, teams, work items, web sites and wiki pages, comments on work items, exception stack traces and test cases.

To enable developers to answer questions that require multiple kinds of information, we developed the information

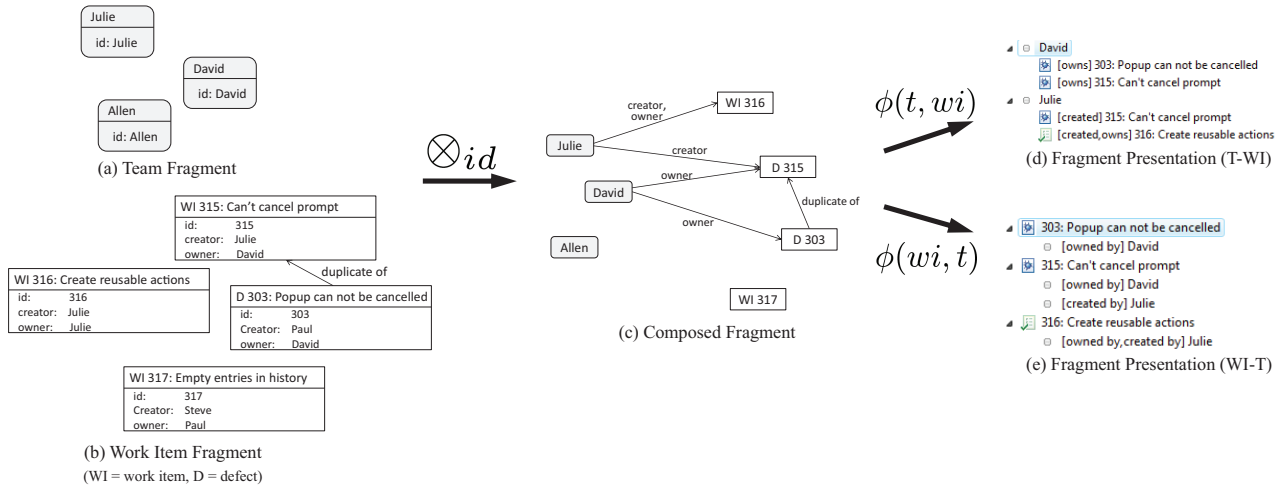

Figure 1: Using the Information Fragments Model to Answer "What have people been working on?"

fragment model that supports the automatic integration of different kinds of information using the structure of the information. A developer can indicate which portions of information in the development environment to integrate, which we refer to as *information fragments*, and can adapt the *presentation* of resultant integrated information to support answering the questions according to personal preferences. The integration of the information, which we refer to as *composition* is done automatically.

Consider the question "What have people been working on?". To answer this question, with the model, a developer can select two information fragments: one comprising the team on which the developer works (Figure 1(a)), and one comprising the work items on which people have been working on (Figure 1(b)). In our approach, information fragments are modeled as graphs with nodes and edges. Nodes represent uniquely identifiable items with properties. For example, a work item includes an identifier (id), a creator and an owner property. Edges represent relationships between items, such as a "duplicate-of" relationship between two work items. The developer then drags and drops these fragments into a special view supporting the model and the fragments get automatically composed, in this case with the identifier matching composition operator $\otimes_{id}$. The composition operator creates a new information fragment (Figure 1(c)) based on the input fragments with new edges introduced between nodes based on the nodes' properties. For example, in the new information fragment, a new "owner" edge is created between David and the defect 303, because the owner property of the node representing defect 303 matches the identifier of the node representing David.

As the answer to the question, one developer likes to see the work items ordered by developers and therefore chooses a presentation that orders nodes of the team fragment (t) above the ones from the work item fragment (wi); this presentation is referred to as a projection ($\phi(t, wi)$). Figure 1(d) shows the result of this projection. In this presentation, the teammate Allen does not show up in the final presentations as he did not work on, and is thus not connected to, any work item in the fragment. Alternatively, a developer might be interested in seeing her team members in context of the work items on which they have been working. In this case, she would use a projection $\phi(wi, t)$. This projection shows the work items her fellow team members have been working on first and the developers below (Figure 1(e)). By separating the presentation from the composition of the information, either interpretation of the question is easily supported.

3.4 Information Fragments Model Evaluation

Using our model, we have been able to express all 78 questions determined through the interview-based study with the 11 professional developers. Just because a model is expressive does not mean it is usable. To investigate whether developers can use the model easily and effectively to answer their questions across multiple domains of information, we conducted a case study with 18 professional developers. To allow the investigation of the usefulness of the information fragments model, we implemented a prototype supporting the model within an integrated development environment. In the case study, the 18 developers were able to easily apply the model to successfully answer a high percentage (94%) of questions posed in minimal time with little training. The information fragment model is an approach to balance expressivity with simplicity. While it is less expressive than general query languages such as SQL [3, 5], it still provides the flexibility to answer all 78 questions according to the information needs of the developers and it supports the questions by automatically composing the information fragments easing the composition of the information. The automatic composition allows a developer to avoid specifying how different information fragments have to be linked. The separation of composition from presentation in the information fragments model allows the developers to tailor the composed information to their personal needs. In the study, we found that developers used the model in different ways to answer the same question, suggesting that the approach supports individual preferences.

3.5 Using Knowledge and Context for Awareness

The degree-of-knowledge and the information fragments model can be synergistically used together, for instance, for project awareness. To stay aware of relevant information in a project, developers often subscribe to information streams,

such as RSS feeds. However, not all of the information in a stream is relevant. Finding the relevant information within the vast amount of information a developer faces each day is difficult, in particular, since the information is presented without context and the developer has to manually relate it to her work. The case study on identifying interesting change sets presented earlier already provides evidence that the DOK model can be used to identify relevant information. In addition, by supporting the composition of fragments of various kinds of information, the information fragments model can provide context that eases the task of staying aware for developers. For instance, using the developer's workspace as a context to rank and sort the information in a stream, might allow the developer to quickly determine which items in the information stream affect or reference the code in the workspace that are of relevance to him. By combining the DOK model with the information fragment model we might be able to provide relevant information in context to support a developer in staying aware of feeds. In an exploratory study with five professional developers, we found initial evidence that the two models can be synergistically combined to support a developer in staying aware, but further study is needed to support our hypothesis.

4. RESULTS AND CONTRIBUTIONS

During a workday, a software developer must continuously search for the small portions of information pertinent to her work within the flood of project information. Today's artifact-centered development environments make finding the needed information tedious or infeasible. In our research, we introduced two models, the degree-of-knowledge model and the information fragments model. These two models show that it is possible to add developer-centric models to a development environment and ease a developer's access to the information relevant to work-at-hand addressing the developer's individual information needs.

The degree-of-knowledge (DOK) model provides a means of describing which part of a code base each developer knows. This model takes an individual perspective on a developer's knowledge of code and captures the ebb-and-flow of the developer's knowledge. The information fragments model provides a means for a developer to integrate information from multiple sources according to the developer's personal preferences. Our research makes the following contributions with respect to the two developer-centric models. For the degree-of-knowledge model:

- we show empirically that a developer's interaction with code can indicate her knowledge and enumerate additional factors that affect a developer's knowledge of code;
- we present evidence that shows a developer's authorship of and interaction with code capture different aspects of knowledge about code;
- we introduce the degree-of-knowledge (DOK) model, an individual view of a developer's knowledge of code;
- we report on the use of the DOK model in three scenarios, discussing benefits and limitations of the model; and
- we discuss the robustness of the model across a variety of development teams.

For the information fragment model:

- we identify and present 78 questions that developers ask and how the developers desire these questions to be answered;
- we introduce the information fragment model and show that it is sufficiently expressive to answer all 78 questions;
- we demonstrate that developers can successfully apply the model through a study with 18 professional developers.

We have discussed how these two models might even be synergistically combined to support a developer in staying aware of relevant information in a project, helping her to rank and filter the new information in her working context. Directions for future work lie in applying both models to other scenarios as well as other kinds of software artifacts.

5. REFERENCES

[1] R. R. Burton, L. M. Masinter, D. G. Bobrow, W. S. Haugeland, R. M. Kaplan, and B. A. Sheil. Overview and status of doradolisp. In *Proc. of LFP'80*.

[2] R. H. Campbell and P. A. Kirslis. The saga project: A system for software development. In *Proc. of SDE'84*.

[3] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proc. of SIGFIDET'74*.

[4] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proc. of CHI'07*.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 1970.

[6] R. DeLine and K. Rowan. Code canvas: zooming towards better development environments. In *Proc. of ICSE '10*.

[7] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proc. of CSCW '92*.

[8] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 1992.

[9] T. Fritz. Staying aware of relevant feeds in context. In *Proc. of ICSE '10*.

[10] T. Fritz and G. C. Murphy. Determining relevancy: How software developers determine relevant information in feeds. Technical report. to appear.

[11] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proc. of ICSE'10*.

[12] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proc. of ESEC-FSE'07*.

[13] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proc. of ICSE'10*.

[14] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *Proc. of CSCW'04*.

[15] D. Jin and J. R. Cordy. Ontology-based software analysis and reengineering tool integration: The oasis service-sharing methodology. In *Proc. of ICSM '05*.

[16] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of FSE'06*.

[17] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with isparql and a software evolution ontology. In *Proc. of ICSEW'07*.

[18] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. of ICSE'07*.

[19] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. of ICSE'06*.

[20] D. W. McDonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proc. of CSCW '00*.

[21] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proc. of ICSE'02*.

[22] S. Paul and A. Prakash. A query algebra for program databases. *IEEE Trans. Softw. Eng.*, 1996.

[23] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proc. of ICSE'03*.

[24] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proc. of CASCON '97*.

[25] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 2005.

[26] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proc. of ICSE'94*.