Computing Degree-of-Knowledge Values for a Developer’s Workspace

by

Jingwen Ou

B.Sc., Guangdong University of Technology, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)
May 2009
© Jingwen Ou, 2009
Abstract

Previous research in computer science shows that software developers are typically deluged by an enormous volume of information daily. Improving the effectiveness of developers to filter this information may yield significant productivity improvements. To combat this overload, we introduce an indicator, called degree-of-knowledge (DOK), which is a real value indicating how much knowledge a developer has with a source code element. A developer's DOK values for a source code base can be computed automatically from authorship data mined from the source revision systems and from interaction data collected as the developer works. This indicator may help reduce information overload by, for instance, filtering the source code to only show the elements for which a developer has high knowledge. We describe our implementation of an efficient framework for computing DOK values in a development environment.
Table of Contents

Abstract .................................................. ii

Table of Contents ........................................ iii

List of Tables ............................................... v

List of Figures .............................................. vi

List of Programs ........................................... vii

Acknowledgements ......................................... 1

1 Introduction .............................................. 2

2 Related Work ............................................. 5
  2.1 Expertise Recommenders ............................... 5
  2.2 Degree-of-Interest Model ............................. 6
  2.3 Indicators of Knowledge of Code ...................... 7
  2.4 Awareness Approaches ................................. 7

3 Model ..................................................... 8
  3.1 The Components of Degree-of-Knowledge .............. 8
    3.1.1 Degree-of-Interest ................................. 8
    3.1.2 Degree-of-Authorship ............................. 9
    3.1.3 Degree-of-Knowledge .............................. 9
  3.2 Events Used to Compute Degree-of-Knowledge .......... 10
    3.2.1 Interaction Events ................................. 10
    3.2.2 Authorship Events ................................. 12
  3.3 Computing Degree-of-Knowledge ...................... 13
    3.3.1 Degree-of-Interest ................................. 13
    3.3.2 Degree-of-Authorship ............................. 16
    3.3.3 Degree-of-Knowledge .............................. 18
# Table of Contents

4 Implementation .................................................. 19
   4.1 Architecture ........................................... 19
   4.2 Core .................................................. 21
   4.3 Storage of Knowledge .................................. 23
   4.4 Monitor ................................................. 24
      4.4.1 Interaction Monitoring ........................... 25
      4.4.2 Authorship Monitoring ............................ 27
   4.5 Extensions ............................................... 33
      4.5.1 Bridging to a Source Revision System .......... 33
      4.5.2 Adding a Knowledge Indicator .................. 33
   4.6 Performance ............................................. 34
      4.6.1 One Working Day Scenario ......................... 35
      4.6.2 Seven Working Days Scenario ..................... 35

5 Discussion ...................................................... 39
   5.1 Scoped Structured View ................................ 39
   5.2 Awareness of Knowledge ................................ 40
   5.3 Knowledge Map .......................................... 43
   5.4 Change Set Assessment .................................. 43
   5.5 Onboarding ............................................. 44
   5.6 Future Work ............................................. 44
      5.6.1 Adding More Indicators ............................ 44
      5.6.2 An Automated Learning Model ..................... 45

6 Summary .......................................................... 46

Bibliography ...................................................... 47
List of Tables

3.1 Indicator event schema ...................................................... 10
3.2 Interaction event types .................................................... 11
3.3 Sample interaction history ............................................... 11
3.4 Authorship event types .................................................. 12
3.5 Sample authorship history ............................................... 13
3.6 Interaction scaling factors ............................................... 16
3.7 Authorship scaling factors .............................................. 17
4.1 XML schema for the persistency of source code elements ...... 24
4.2 Comparison of terms between Jazz Team Server and Concur-
    rent Version System ..................................................... 28
4.3 Number of elements with positive DOK values ................. 37
List of Figures

4.1 Framework architecture showing OSGI plug-ins and their dependencies .......................... 20
4.2 Core plug-in dependency ........................................................................... 21
4.3 Event-based model ................................................................................. 22
4.4 Knowledge model class diagram .............................................................. 23
4.5 Event-based model class diagram .............................................................. 25
4.6 Interaction monitor plug-in dependency ..................................................... 26
4.7 Jazz Team Server monitor plug-in dependency .......................................... 30
4.8 Concurrent Versions System monitor plug-in dependency ......................... 32
4.9 Repository bridge class diagram ............................................................... 34
4.10 Used Java heap in kilobytes for one simulated workday .......................... 36
4.11 CPU time in milliseconds for one simulated workday .............................. 36
4.12 Used Java heap in kilobytes for seven simulated workdays .................. 38
4.13 CPU time in milliseconds for seven simulated workdays ....................... 38
5.1 Scoping task context with DOK model .................................................... 40
5.2 Awareness of knowledge drop when synchronizing workspace ............... 42
5.3 Part of a knowledge map ......................................................................... 43
List of Programs

3.1 Incremental DOI Computation .................. 15
3.2 Incremental DOA Computation .................. 17
Acknowledgements

I thank my supervisor Gail C. Murphy for her guidance, support and encouragement that make this work possible. Thanks to her for being supportive for me to work on the thesis. Thanks to her for all the insightful discussions that will continue to guide my whole career life.

I thank Gregor Kiczales for being my second reader and giving me invaluable comments.

Thanks to all the great friends I’ve met in Vancouver for making it such an amazing experience.
Chapter 1

Introduction

Software developers use integrated development environments (IDEs) to ease their work on large systems. These environments provide almost immediate access to multiple sources of information about the software under development, including source code, bug reports, and Really Simple Syndication (RSS) feeds. These environments make it easy for a developer to navigate across related pieces of information. For example, most environments make it easy to navigate across calling relationships between source code elements and between historical revisions to elements. Source code elements include types, methods, and fields in a source code base.

These environments have been engineered to provide access to millions of lines of source code. However, in providing access to a large amount of information, these environments are also deluging developers with more information than they need to solve a particular available task, which typically requires only a fraction of the information [9]. Further complicating this problem is that the information accessed by developers is often being changed by other members of a team. A source code element that a developer studied in depth a month ago may be completely changed by the time the developer revisits the element. In a recent study, it was shown that for a group of industrial developers over a three-month period, an element accessible in their development environment was being changed on average once every 54 seconds [7].

The only effort of which we are aware to directly address the information overload problem is the Eclipse Mylyn project. Mylyn is based on the idea that a developer's interaction with a system can be transformed into a degree-of-interest model [11], where each source code element in the system is weighted according to its relevance to the task at hand. For each task, a task context can be formed comprising the interactions a developer has had with the source while working on the task. From the information in a task context, the set of source code elements with a positive degree-of-interest can be constructed. In this way, the task context can be used to

\[1\text{http://www.eclipse.org/mylyn/}, \text{verified 25 April 2009}\]
Chapter 1. Introduction

focus the User Interface (UI) of the development environment by highlighting the most important elements, filtering the unimportant ones, and allowing developers to perform operations, such as performing check-ins to a source code repository, on elements of interest.

The main attempt to address problems related to information change in a development environment has been approaches to make developers aware of changes as they are occurring. Palantir [1], for example, increases a developer's awareness by continuously sharing information regarding the activities of other developers, instead of informing developers of other efforts only when they themselves perform some configuration management operation (e.g., check in or check out). Specifically, Palantir informs a developer of which other developers change which other artifacts, calculates a simple measure of severity of those changes, and graphically displays the information in a configurable and generally non-obtrusive manner.

In this thesis, we introduce an approach that has the potential to address both the overload and the high rate of change problem. We introduce an indicator for which code in an environment a developer has knowledge about. With such an indicator, information overload may be reduced by showing only elements for which a developer has high knowledge for the past week. This filtering might help reduce information overload by helping to seed appropriate elements into the start of a programming task, which can then be managed by Mylyn. With such an indicator, a developer could potentially address issues related to the high rate of change of information in the environment. For instance, when receiving a group of source revisions, for example a change set, from a team member that is intended to fix a bug or implement an enhancement, a developer must assess the impact of the change set: will the revisions integrate easily or will they cause the system to break? An indicator of which elements related to the change set a developer has good knowledge of might help a developer to focus their attention on the change sets and code that need the most review.

A previous study conducted about a developer's knowledge of code [6] suggests that two important components that indicate a developer's knowledge are the interactions a developer has had with the code as captured by a degree-of-interest model [11], and which code the developer has authored. The knowledge indicator we introduce in this thesis, a degree-of-knowledge (DOK) value, incorporates both of these components. A DOK value for a source code element is specific to a developer and can be automatically computed from data gathered about and during the development process (Section 3). Different developers may have different DOK values for the same source code elements. We compute the DOK value for a developer by
combining authorship data from the source revision system and interaction data from monitoring the developer's activity in the development environment. In addition to presenting an abstract formulation of the DOK model (Section 3), we also describe an efficient implementation to compute DOK values as a developer works with a large amount of code in a development environment (Section 4).

The main focus of this thesis is the definition and implementation of an efficient framework for computing DOK values in a development environment. While experimentation to determine the usefulness of DOK values to address the problems outlined above is future work, we provide a glimpse of the kinds of information a developer might find useful, using the DOK values (Section 5).

We make the following contributions in this thesis.

- We provide a model for a degree-of-knowledge (DOK) value that incorporates both a developer's interaction with code elements and a developer's authorship of code elements.

- We describe an efficient and extensible implementation for the DOK model.

- We discuss the usage of the DOK model in different scenarios, reporting on the benefits of the DOK model.
Chapter 2

Related Work

We build our DOK model by analyzing authorship information from a project's source revision system and interaction information from an integrated development environment (IDE). In this chapter, we present work related to building a knowledge model, including expertise recommender tools that mine authorship change information from source repositories (Section 2.1) and an approach that approximates knowledge per task based on interaction information (Section 2.2). We also describe a user study that suggests significant factors in building a DOK model (Section 2.3) and compare our approach to awareness tools (Section 2.4).

2.1 Expertise Recommenders

Earlier work has considered how to determine which developers are experts in, or have knowledge of, particular parts of a source code base by relying on authorship change information from a project's source repository (e.g., [14]). Most of these approaches to constructing a knowledge model are based on a heuristic called the "Line 10 Rule", which states that the developer who last committed a change to a file has expertise in that file [12]. One of the earliest systems, the Expertise Recommender, selects an expert assuming that the developer who made the most recent change to a source file has the relevant expertise [12]. Another system, the Expertise Browser, uses experience atoms, basic units of experience created by mining the version control system, to rank developers according to revision times so that the developer who made the last change to the source file has the highest rank of knowledge on a particular part of the system [14]. The Emergent Expertise Locator goes one step further by considering the relationship between file modifications and who made those changes [13]. Girba and colleagues used finer-grained information by considering the number of lines of code that each developer has modified when equating expertise [8].

All of these previous approaches consider a developer's expertise either as a binary function or as a monotonically increasing function. The Expertise Recommender considers the developer who made the last change to a file
2.2 Degree-of-Interest Model

as the only one having expertise in that file. The Expertise Browser and Emergent Expertise Locator rank expertise by accumulating information of who made recent changes to a file, without taking care of the fact that acceptance of changes to a file by other developers also likely degrades the developer's expertise in this file.

Our approach also considers authorship information from a source project's repository. However, we refine the existing approaches by modeling the ebb and flow of multiple developers changing the same file: a developer's degree-of-knowledge in the file rises when the developer commits changes to the source repository and diminishes when other developers make subsequent changes to the same code. Our approach also considers the interactions the developer has with the source codebase in this ebb and flow: a developer's degree-of-knowledge in the file rises when the developer interacts with it and diminishes when she interacts with other files.

2.2 Degree-of-Interest Model

Another way to gain knowledge about source code is to interact with the code in a development environment. Kersten et al. used this type of approach, proposing a degree-of-interest (DOI) value to represent which program elements a developer has interacted with significantly [9]. The basic idea is that the more frequently and recently a developer has interacted with a particular program element, the higher the DOI value; as a developer moves to work on other program elements, the DOI value of the initial element decays. The original applications of this concept computed DOIs across a developer's workspace and filtered the views in the development environment based on DOI values to reduce information overload [10]. Subsequent work focused the DOI computation on a per-task basis since a developer often works on and switches between tasks and computing DOI values across the whole workspace did not adequately scope the display of appropriate element [11]. In this thesis, we refine the approach by considering how a developer interacts with the code in a development environment as they collaborate with teammates by producing changes to the code. We chose to return to the computation of DOI across all of a developer's workspace to capture a developer's knowledge of the source across tasks.
2.3 Indicators of Knowledge of Code

Fritz et al. has conducted an experiment to investigate what factors could indicate for which code a developer has knowledge [6]. Through this study, it was found that DOI values computed from the interaction information can indicate knowledge about a developer's code and other factors, such as authorship of code, also play a significant role in gauging a developer's knowledge of the code. Our DOK model uses the results of this previous study by considering two key factors that impact a developer's knowledge of code: interaction and authorship. The focus of this thesis is the implementation of the DOK model. An investigation of whether such a model can help with the information overload problem is reported on elsewhere [7].

2.4 Awareness Approaches

Various awareness tools have been proposed for collaborative development environments to make developers aware of other team members' work. [3]. For example, Flesce [2] features real-time awareness of team activities by providing a shared environment (e.g., a shared debugger); Palantir [1] enables developers to detect potential conflicts early, as they occur, by providing workspace awareness in the form of which developers are changing which artifacts and by how much. Elvin [5], an augmentation to CVS with real-time notification and chat facilities, sends real-time CVS log messages to developers on what code changes are happening and enables them to engage in a timely work discussion.

These existing approaches focus on the immediate presentation of changes to artifacts from various sources (e.g., source control system). In contrast, our tool builds a model of that aims to raise developers' awareness of important information in the workspace weighted by the DOK values. For example, our tool might be used to filter the source code elements that a developer has knowledge of and focus her attention on the elements that she potentially needs to learn (e.g., unfamiliar elements that are related to a bug).
Chapter 3

Model

Our degree-of-knowledge (DOK) model captures an individual developer's perspective on her source code. Each source code element loaded into a development environment used by a developer is assigned a degree-of-knowledge value. The intent of the model is the higher the value, the more familiar the developer is with the source code element.

Our model captures a DOK value for a source code element based on one component that indicates a developer's short-term knowledge of a source code element, represented by a degree-of-interest (DOI) value, and a second component that indicates a developer's longer-term knowledge of the element, represented by a degree-of-authorship (DOA) value. To better describe our concepts, we use D1 to represent the developer for whom DOK values in source code elements are being computed.

3.1 The Components of Degree-of-Knowledge

A developer works on source code and builds up DOK values for that source code over long periods of time. As the developer works, the integrated development environment (IDE) used by the developer, is started and stopped. We refer to the periods during which the IDE is active as a session. For instance, suppose D1 works 8 hours one day and at the end of the workday exits the IDE, forming one session. She then invokes a new session the next workday. To accurately represent DOK values, we must compute DOI and DOA values across these sessions. When describing how we compute DOK values, we also use the term workspace, which refers to the source code elements a developer has access to in the IDE.

3.1.1 Degree-of-Interest

The intent of DOI is to capture the interest level of D1 in a source code element on which D1 is working. The DOI concept was developed to represent a developer's interest in an element as part of working on a task [9]. In this work, we compute DOI over all of D1's interactions with the source code,
3.1. The Components of Degree-of-Knowledge

similar to the original calculation of DOI [10]. We compute DOI in this way because the scope of a developer's knowledge is all source code elements in her workspace instead of elements specific to a task.

The DOI value for an element is based on the frequency and recency of interactions that D1 has with the source code element. The frequency is determined by the number of interactions with the element as a target. The recency is defined by a decay that is proportional to the total number of interactions a developer has with the elements in the workspace and the first interaction with the element.

3.1.2 Degree-of-Authorship

The DOA component is intended to represent how well D1 knows a source code element based on whether or not she wrote code comprising that element. In a team development situation, it is possible (and even likely) that over time, other developers also contribute to the same source code element as D1. Contributions by others to the element are assumed to decrease D1's familiarity with the element [6]. The DOA for D1 in a source code element can rise and fall as D1 and others modify the element.

We consider the DOA of D1 of an element to be determined by three factors:

- first authorship (FA), representing whether D1 created the first version of the element;
- the number of deliveries (DL), representing subsequent changes by D1 to the element after initial authorship; and
- the number of acceptances (AC), representing changes to the element not contributed by D1.

3.1.3 Degree-of-Knowledge

The DOK represents the knowledge of D1 over a source codebase. It is a linear combination of the DOA and DOI of a source code element e:

\[ DOK(e) = \alpha \times FA(e) + \beta \times DL(e) + \gamma \times AC(e) + \delta \times DOI(e) \] (3.1)

To determine appropriate weightings (\(\alpha, \beta, \gamma, \delta\)), an experiment was conducted with six professional Java developers [7].
3.2. Events Used to Compute Degree-of-Knowledge

Computing DOK requires information about how a developer is working with source code elements in a development environment. At the heart of the model are the events that encapsulate the activities of a developer in an IDE. We use the term indicator event to represent events monitored in a development environment to help compute DOK. Two kinds of indicator events are used:

1. interaction events, representing how a developer selects, edits and otherwise interacts with source code elements in her workspace, and
2. authorship events, representing changes made to an element and the author of those changes.

Each indicator event captures at least four pieces of information (Table 3.1). The time field stamps an event with the moment that the event occurred. The structure kind field provides an identifier that binds the element to a particular domain structure available within the IDE (e.g., "java"). The structure handle field provides an identifier that uniquely identifies the element for the given structure type within the IDE. The event type field describes the type of the event. The contribution value field indicates how much the event contributes to the computation of the DOK value.

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>The time of the event occurrence.</td>
</tr>
<tr>
<td>Structure Kind</td>
<td>Identifier describing the kind of element operated upon.</td>
</tr>
<tr>
<td>Structure Handle</td>
<td>Identifier of the target element.</td>
</tr>
<tr>
<td>Event Type</td>
<td>The type of the event.</td>
</tr>
<tr>
<td>Contribution Value</td>
<td>The value contributing to the computation of the DOK value.</td>
</tr>
</tbody>
</table>

3.2.1 Interaction Events

As a developer works, she interacts with the source code through the development environment. These interactions can be captured and are referred to as interaction events. Interaction events are one kind of indicator event.
3.2. Events Used to Compute Degree-of-Knowledge

Some interaction events are the result of a developer’s interactions with source code elements. For instance, a developer D1 may select a particular Java method to view its source. D1 may then edit the element before saving the file containing the element. Each of these interactions corresponds to an event of a different kind carrying different contribution values. Some interaction events are a summary of interaction events in previous sessions, for instance, the system remembers the elements that D1 interacted with in previous sessions and reproduces the elements’ DOI values next time she starts up the IDE. Table 3.2 summarizes the three supported interaction event types.

<table>
<thead>
<tr>
<th>Category</th>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interaction</td>
<td>SELECTION</td>
<td>Selections of an element via mouse or keyboard in any editor or view.</td>
</tr>
<tr>
<td></td>
<td>EDIT</td>
<td>Edit of an element in any editor.</td>
</tr>
<tr>
<td>system generated</td>
<td>INITIAL.DOI</td>
<td>Reproduces an element’s DOI value of previous sessions.</td>
</tr>
</tbody>
</table>

Table 3.3: Sample interaction history

<table>
<thead>
<tr>
<th>Event</th>
<th>Kind</th>
<th>Target(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INITIAL.DOI</td>
<td>ContextStructureBridge.getHandleIdentifier() method</td>
</tr>
<tr>
<td>2</td>
<td>SELECTION</td>
<td>ContextStructureBridge class</td>
</tr>
<tr>
<td>3</td>
<td>EDIT</td>
<td>ContextStructureBridge.getHandleIdentifier() method</td>
</tr>
<tr>
<td>4...10</td>
<td>EDIT</td>
<td>ContextStructureBridge.getMethodContentType() method</td>
</tr>
<tr>
<td>11</td>
<td>SELECTION</td>
<td>ContextStructureBridge.getHandleIdentifier() method</td>
</tr>
</tbody>
</table>

Table 3.3 provides an example of the sequence of interaction events resulting from part of D1’s work session. For simplicity, we use an event number to stand in for the time field of an interaction event. D1 first starts up the IDE (event 1 in Table 3.3) which causes the DOI values of all source code elements that D1 knows from previous sessions to be loaded. Then, she selects a class, ContextStructureBridge (event 2) and invokes a rename operation on one of its methods (event 3). D1 then edits another method in the same class (event 4 to 10) before she selects the method again (event
3.2. Events Used to Compute Degree-of-Knowledge

11) to facilitate subsequent modifications (not shown).

3.2.2 Authorship Events

As a developer works in a team, she interacts with other developers by contributing to the same source code base through the development environment. A developer can push or pull changes to and from the source revision system, typically through a shared team view in the IDE. We refer to the operations captured from this view as authorship events.

Some authorship events are the result of the changes made to a program element and shared with the team; for instance, a developer D1 is the first author of a particular Java method and makes three changes to it before accepting a change from another developer D2. Each of these authorship changes corresponds to an event of a different kind with different contribution values. Only changes that are shared with the team are considered as authorship events; changes made by a developer local to their workspace are not considered. Other authorship events are also generated by the system to summarize the authorship events in previous sessions. Four authorship event types are supported as summarized in Table 3.4.

<table>
<thead>
<tr>
<th>Category</th>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authorship change</td>
<td>FIRST_AUTHORSHIP</td>
<td>The creation of the first version of an element.</td>
</tr>
<tr>
<td></td>
<td>DELIVERY</td>
<td>Subsequent changes after first authorship made to the element by a developer and shared with the team.</td>
</tr>
<tr>
<td></td>
<td>ACCEPTANCE</td>
<td>Changes to the element made by other members of the team and shared with the developer for whom DOA is being calculated.</td>
</tr>
<tr>
<td>system generated</td>
<td>INITIAL_DOA</td>
<td>Accesses an element's DOA value of a previous session.</td>
</tr>
</tbody>
</table>

Table 3.5 provides an example of part of a sequence of authorship events of one of D1's work sessions. As before, we use an event number to stand in for the time field of an authorship event. D1 first starts up the IDE.
which causes the DOI values of all source code elements that D₁ knows from previous sessions to be loaded (event 1 in Table 3.5). Then, she adds a method to a class and delivers the changes (event 2). D₁ then makes and delivers three changes to the method (event 3 to 5)) before she accepts another new change to the method (event 6) by her teammate D₂.

Table 3.5: Sample authorship history

<table>
<thead>
<tr>
<th>Event</th>
<th>Kind</th>
<th>Target(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INITIAL_DOA</td>
<td>CortanaCore class</td>
</tr>
<tr>
<td>2</td>
<td>FIRST_AUTHORSHIP</td>
<td>CortanaCore.getStructureBridge() method</td>
</tr>
<tr>
<td>3...5</td>
<td>DELIVERY</td>
<td>CortanaCore.getStructureBridge() method</td>
</tr>
<tr>
<td>6</td>
<td>ACCEPTANCE</td>
<td>CortanaCore.getStructureBridge() method</td>
</tr>
</tbody>
</table>

3.3 Computing Degree-of-Knowledge

Each event type has a different scaling factor constant, resulting in different weightings for different kinds of interactions and authorship changes. We conducted an experiment to empirically investigate these weightings [7]. In essence, the experiment involved gathering data about authorship from the revision history of a project and about interest by monitoring developers' interactions with the code as they work on the project. We then asked the developers to rate their knowledge in particular code elements. With the developer ratings, we used multiple linear regression to determine appropriate weightings for the various factors described in Section 3.1.3.

To support DOK computation over a long period of time, we use incremental algorithms. Starting with the DOK values provided from a previous session, the incremental algorithms continuously compute the knowledge that a developer has with the source code elements.

3.3.1 Degree-of-Interest

We compute DOI for an element using the basic approach used in the Eclipse Mylyn project, based on frequency and recency of a developer's interactions with the element. However, unlike Mylyn, there is a need to compute the values across sessions and for a larger number of source code elements at
3.3. Computing Degree-of-Knowledge

once, since we need to compute DOK over all elements in the workspace. The DOI value of a target element $e$ is calculated by dividing it into sessions:

$$\text{DOI}(e) = \sum_{i=0}^{n} \text{DOI}_{\text{session}}(e)_i$$

(3.2)

where

$$\text{DOI}_{\text{session}}(e) = \text{scaling}_{\text{selection}} \times \text{selections}_{\text{session}}(e)$$

$$+ \text{scaling}_{\text{edit}} \times \text{edits}_{\text{session}}(e)$$

$$- \text{scaling}_{\text{decay}} \times \text{event\_distance}_{\text{session}}(e)$$

(3.3)

$\text{DOI}_{\text{session}}(e)$ is the sessional DOI value of the target element $e$, $\text{selections}_{\text{session}}(e)$ is the number of selections of $e$ in the session, $\text{edits}_{\text{session}}(e)$ is the number of edits of $e$ in the session, and $\text{event\_distance}_{\text{session}}(e)$ is the position in the sessional event stream of the first interaction with $e$.

Program 3.1 describes how we incrementally compute a DOI value for an element, given an interaction event with the element as the target. The intent of the PROCESS\_EVENT function is to compute the new DOI value for an element when an interaction event occurs and to ensure that the DOI value of an element becomes positive if the DOI value for the element has decayed to a negative interest when a new event occurs. The definition of PROCESS\_EVENT uses a data structure, called knowledgeElement, to store the decay start event number (knowledgeElement.decayStart) and the interest value (knowledgeElement.interest) of the element (Section 4.2).

In PROCESS\_EVENT function, we retrieve the previous interest value of the element (line 3), increment the interest value of the element based on the kind of the current event (line 4) and, if the decay has offset the interest, reset the decay to start at the current event (line 7-8). Finally the interest value of the element is updated (line 9). The GET\_KNOWLEDGE\_ELEMENT function returns or creates the data structure (knowledgeElement) of the target element. The DECAY function computes the decay based on the distance of two interaction events, where the distance is in terms of the number of events instead of the time stamp of the events. The DOI function computes the real degree-of-interest value of an element. The LAST\_EVENT is a counter that counts the total number of interaction events. The SCALING function returns the constant associated with each event kind and with the KIND\_DECAY constant.

The scaling factors used are the same as in the Eclipse Mylyn project and are summarized in Table 3.6.
3.3. Computing Degree-of-Knowledge

Program 3.1 Incremental DOI Computation

```
PROCESS_EVENT(element, elementEvent)
1   knowledgeElement = GET_KNOWLEDGE_ELEMENT(element)
2   decayStart = knowledgeElement.decayStart
3   interest = knowledgeElement.interest
4   interest += SCALING(KIND(elementEvent))
5   currDecay = DECAY(decayStart, elementEvent)
6   if interest < currDecay then    // reset decay & interest
7       knowledgeElement.decayStart = elementEvent
8       interest = SCALING(KIND(elementEvent))
9       knowledgeElement.interest = interest

DECAY(fromEvent, toEvent)
10  decayDistance = |toEvent - fromEvent|
11  return decayDistance * SCALING(KIND-DECAY)

DOI(element)
12  knowledgeElement = GET_KNOWLEDGE_ELEMENT(element)
13  decayStart = knowledgeElement.decayStart
14  interest = knowledgeElement.interest
15  totalDecay = DECAY(decayStart, LAST_EVENT)
16  return interest - totalDecay
```
3.3. Computing Degree-of-Knowledge

Table 3.6: Interaction scaling factors

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Scaling Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECTION</td>
<td>1</td>
</tr>
<tr>
<td>EDIT</td>
<td>0.7</td>
</tr>
<tr>
<td>DECAY</td>
<td>0.017</td>
</tr>
</tbody>
</table>

We use the events in Table 3.3 to illustrate the algorithm. To simplify the example, we assume the scaling factors for SELECTION and EDIT are both 1 and the scaling factor for DECAY is 0.5. We compute the DOI value for the ContextStructureBridge.getHandleIdentifier method. It has an initial DOI value of 2 from previous sessions (event 1 in 3.3). Then there is a selection on another element so its DOI value decays by 0.5 and drops to 1.5 (event 2). It is then edited and has the DOI value of 2.5 (event 3). From event 4 to 10, it decays by 3 in total and its DOI value drops to -0.5. Then it is selected again (event 11) and its decay value and interest value are reset hence having the final DOI value of 1.

3.3.2 Degree-of-Authorship

We compute DOA based on the frequency of authorship change, which is determined by three event types: first authorship, the number of deliveries and the number of acceptances. Each event type has a different scaling factor constant, resulting in different weightings for different kinds of authorship changes. We also calculate the DOA value by dividing it into sessions and the DOA of a target element \( e \) for a developer \( D_1 \) can be defined as

\[
DOA(e) = \sum_{i=0}^{n} DOA_{session}(e)_i
\]

where

\[
DOA_{session}(e) = scaling_{first\_authorship} \times first\_authorship_{session}(e) \\
+ scaling_{delivery} \times deliveries_{session}(e) \\
- scaling_{acceptance} \times acceptances_{session}(e)
\] (3.5)

\( DOA_{session}(e) \) is the sessional DOA value of the element \( e \), \( first\_authorship \) is whether the developer \( D_1 \) created the first version of \( e \) in the session, \( deliveries \) is the number of changes made to \( e \) and delivered by \( D_1 \), and
3.3. Computing Degree-of-Knowledge

acceptances is the number of changes made to $e$ not by $D_1$ but accepted by $D_1$.

Program 3.2 describes how we incrementally compute a DOA value for an element, given an authorship event with the element as the target. We get the previous authorship value (line 2), increment the DOA value of the element on the kind of the current event (line 3), and finally update the authorship value of the element (line 4).

**Program 3.2 Incremental DOA Computation**

```python
PROCESS_EVENT(element, elementEvent)
1   knowledgeElement = GET_KNOWLEDGE_ELEMENT(element)
2   authorship = knowledgeElement.authorship
3   knowledgeElement += SCALING(KIND(elementEvent))
4   knowledgeElement.authorship = authorship

DOA(element)
5   knowledgeElement = GET_KNOWLEDGE_ELEMENT(element)
6   return knowledgeElement.authorship
```

The SCALING function returns the constant associated with each event kind. These scaling factors come from an empirical experiment [7] and are summarized in Table 3.7.

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Scaling Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST.AUTHORSHIP</td>
<td>1.098</td>
</tr>
<tr>
<td>DELIVERY</td>
<td>0.164</td>
</tr>
<tr>
<td>ACCEPTANCE</td>
<td>-0.321</td>
</tr>
</tbody>
</table>

Using the events from Table 3.5, we compute the DOA value of the CortanaCore.getStructureBridge method for developer $D_1$. To simplify the example, we assume the scaling factors for FIRST.AUTHORSHIP, DELIVERY and ACCEPTANCE are 1, 0.8 and -0.5 respectively. The method does not have any initial DOA value from previous sessions so its DOA value is 0 (event 1 in Table 3.5). Then $D_1$ shares the method that she authored with her teammate and delivers to the source revision system (event 2), hence it has the DOA value of 1. From event 3 to 5, $D_1$ delivers another three revisions of the methods hence having the DOA value of 3.4. Then she accepts
3.3. Computing Degree-of-Knowledge

a change to the method (event 6) from her teammate D₂ hence having the final DOA value of 2.9.

3.3.3 Degree-of-Knowledge

We combine the DOI and DOA of a source code element for a developer to provide an indicator of the developer's knowledge in that element. The degree-of-knowledge we compute linearly combines the factors contributing to the DOA and the DOI. We determined the scaling factors through an experiment described elsewhere [7]. The DOK value of a target element $e$ is computed as:

$$DOK(e) = 1.098 \times FA(e) + 0.164 \times DL(e) - 0.321 \times \ln (1 + AC(e)) + 0.19 \times \ln (1 + DOI(e))$$

(3.6)

$FA(e)$ is the first authorship of the element $e$, $DL(e)$ is the number of deliveries of $e$, $AC(e)$ is the number of acceptances of $e$, and $DOI(e)$ is the DOI value of $e$. 
Chapter 4

Implementation

There are two key challenges to implementing the DOK model: determining the mapping from how a developer interacts with a particular IDE and a particular source revision system to the events described in the knowledge model and being able to bridge to different monitors to track these interactions. We implemented the knowledge model for the Eclipse IDE and we call it Cortana² ³. The Eclipse IDE provides a set of unified APIs for monitoring a developer's interaction. To monitor authorship changes in two source revision systems, we reuse two applications built on top of Eclipse: the Rational Team Concert⁴ and the Eclipse CVS client⁵. Rational Team Concert is a collaborative work environment built on top of Eclipse for developers, architects and project managers with work item, source control, build management, and iteration planning support for the Jazz Team Server. The Eclipse CVS client is a component of Eclipse that can connect to the Concurrent Version System. Jazz Team Server and Eclipse CVS client provide APIs to access repository information in different levels.

In this chapter, we describe the framework architecture that implements the DOK model (Section 4.1); we then move onto the implementation of the knowledge model, including how we persist knowledge information between sessions (Section 4.2 and Section 4.3); we then give a description of the implementation of the monitors (Section 4.4) and adding additional monitors and knowledge indicators to the model (Section 4.5).

4.1 Architecture

The framework comprises three components: core, monitor and integration. The core component provides a model and operations not coupled to any particular platform and suitable for use in server-side applications for

²Cortana is a fictional artificially intelligent (AI) character in the Halo video game series who is capable of hacking into human's mind.
³http://cs.ubc.ca/~jingwen/cortana/, verified 29 April 2009
⁴http://jazz.net/, verified 26 April 2009
⁵http://www.eclipse.org/eclipse/platform-cvs/, verified 26 April 2009
4.1. Architecture

Figure 4.1: Framework architecture showing OSGI plug-ins and their dependencies

embedding in another framework. The core component comprises two sub-components: interaction and authorship, which include the model and operations specific to the calculation of DOI and DOA respectively. The monitor component transforms a developer's interaction with, or authorship change of, the source code elements into indicator events that are processed by the core component. It provides extension points to allow different monitor implementations to contribute indicator events to the calculation of DOK (Section 4.4). The integration component integrates different monitor implementations into the framework (Section 4.5.1). In the current version, the framework bridges to the Eclipse Mylyn project's monitor framework for monitoring the user interactions and implements monitors for Jazz Team Server and Concurrent Version System to track authorship changes.

Figure 4.1 illustrates our component model, with each label corresponding to a component and each box corresponding to a subcomponent belonging to the component. The key property of this architecture is loose coupling of components. Each component is implemented as a plug-in within the Eclipse architecture, to allow such extensibility as the use of different monitor implementations.
4.2. Core

The core component computes DOI, DOA and DOK values for each source code element. It also persists the computed DOK values to support use of the model across a developer's sessions with the IDE. Figure 4.2 shows the dependency structure of the core plug-in, with arrows corresponding to plug-in dependencies. The core plug-in (cs.ubc.ca.spl.cortana.core) depends on the event-based model implemented in the monitor plug-in. The plug-ins in grey represent indirect dependencies of the core plug-in.

The framework utilizes an event-based model and the Eclipse extension.
4.2. Core

Figure 4.3: Event-based model

point mechanism to achieve loose coupling of components. As shown in Figure 4.3, the core plug-in registers itself to all monitors integrated to the monitor plug-in (registerListener()) and gets notified whenever an indicator event is issued by the monitors (handleIndicatorEvent()). The core plug-in then distributes the event to the appropriate event handler that contributes to the calculation of DOK (processIndicatorEvent()). Two event handlers are implemented in the current version: the interaction event handler and the authorship event handler. The core plug-in also provides an extension point for incorporating additional factors when computing DOK values (Section 4.5.2).

Figure 4.4 shows the class diagram of the knowledge model in the core plug-in. A knowledge context (IKnowledgeContext) is formed as a developer works. It is constituted by a set of source code elements that the developer has knowledge (IKnowledgeElement). The DOK value of each element increases and decreases as the developer interacts with, or performs authorship change operations on, the source code base. The computation of the DOK value is encapsulated in a composite indicator (IDegreeOfKnowledge), which is a composition of a group of individual knowledge indicators (IDegreeOfIndicator). As discussed in Section 3.1.3, two indicators are implemented: an indicator for short-term knowledge (IDegreeOfInterest) and an indicator for longer-term knowledge (IDegreeOfAuthorship). IDegreeOfInterest provides operations in support of the DOI computa-
4.3 Storage of Knowledge

Figure 4.4: Knowledge model class diagram

...tion by storing the interest value and the decay start event number (Section 3.3.1). Similarly, IDegreeOfAuthorship supports the DOA computation by storing the authorship value (Section 3.3.2). As discussed in Section 4.5.2, the model can be extended to integrate other indicators verified in future work.

4.3 Storage of Knowledge

The core component also provides facilities for the persistency of source code elements for which a developer has DOK values above a certain thresholds. Table 4.1 describes the XML schema used to store the DOK values for elements.

FA, DL, AC and DOI are the aggregate values from previous sessions.
4.4. Monitor

Table 4.1: XML schema for the persistency of source code elements

<table>
<thead>
<tr>
<th>XML Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StructureHandle</td>
<td>Identifier of the target element.</td>
</tr>
<tr>
<td>StructureKind</td>
<td>Identifier describing the kind of element operated upon.</td>
</tr>
<tr>
<td>FA</td>
<td>The value of first authorship.</td>
</tr>
<tr>
<td>DL</td>
<td>The number of deliveries.</td>
</tr>
<tr>
<td>AC</td>
<td>The number of acceptance.</td>
</tr>
<tr>
<td>DOI</td>
<td>The DOI value.</td>
</tr>
<tr>
<td>Time</td>
<td>The time of the element being last interacted or having the last authorship change.</td>
</tr>
</tbody>
</table>

The application persists these values when the IDE is shut down and restores them when the IDE is started up. The reason that we store the components of DOA computation (FA, DL and AC) but not components of DOI (selections and edits) is to conform to the DOK computation formula described in Section 3.3.3.

4.4 Monitor

The monitor component provides different monitor implementations to be integrated into the framework. Since the framework conforms to an event-based mechanism, all monitor implementations should issue a specific indicator event in order to contribute to the DOK computation life cycle.

Figure 4.5 shows the class diagram of some of the essential interfaces in the monitor component. All contributing monitors should implement the IIndicatorEventSource interface to join the framework monitoring life cycle, which issues IIndicatorEvent whenever an operation contributing to an interaction or authorship change is performed. In the current implementation, two indicator events are supported: interaction events (IInteractionEvent) and authorship events (IAuthorshipEvent). An IIndicatorEvent encapsulates information about the source element on which an interaction or authorship change operation occurred (e.g., structure handle and structure kind) and the contribution value of the operation (e.g., an element is selected once hence the event's contribution value is 1).
4.4. Monitor

4.4.1 Interaction Monitoring

The Interaction Monitor subcomponent, an Eclipse plug-in, provides interaction monitoring. The monitor can be installed separately from the framework and collects information about a user's activity in Eclipse. The monitor plug-in adapts to Eclipse Mylyn's monitor framework to track developers' interactions with the IDE. It accepts listeners to different Eclipse events, including preference changes, perspective changes, window events, selections, and commands invoked through menus or key bindings. The IInteractionEvent interface used by the monitor transforms the interacting information that a developer has with source code elements. Scaling factors are used to determine the bias between different kinds of interactions (e.g., bias of edits vs. selections) as listed in Table 3.6.

Figure 4.6 shows the dependencies of this plug-in. It adapts to the framework's event-based model by reusing the interaction monitoring facilities provided by the Eclipse Mylyn project. The idea is that whenever an interaction is performed, the internal event generated by the Mylyn monitor is translated to the indicator event used in the framework and broadcast.
Figure 4.6: Interaction monitor plug-in dependency
4.4 Monitor

4.4.2 Authorship Monitoring

The Authorship Monitor subcomponent provides authorship change monitoring by bridging to different source revision systems. These monitors can be plugged into the framework to track the delivery and acceptance of any source code element in different source repositories. The basic idea is that the monitor listener for the connections between a developer's local workspace and its remote copy and generate events whenever an operation is detected.

Since different source control systems support different types of commands using different terms, we find it necessary to define unified names for the set of generic operations that are supported by most repositories. For example, delivery means the commit of changes after first authorship made to the element by a developer in our framework. However, this command is referred to as a deliver in Jazz Team Server while in CVS it is called a commit.

Two authorship monitors have been currently implemented: a monitor for the Jazz Team Server (Jazz) and a monitor for the Concurrent Versions System (CVS). Each of them monitors the operations that a developer performs in a team shared view in Eclipse by issuing corresponding indicator events. The operations include delivery and acceptance of changes to and/or from the source revision systems.

The IAuthorshipEvent interface is used by the monitors to encapsulate the authorship information when a developer changes an element, including information on the type of change (e.g., adding a method) and the author who made the change. Scaling factors listed in Table 3.7 are used to determine the bias between different kinds of authorship changes (e.g., bias of first authorship vs. acceptance).

Before going deep into the discussion of the implementation, we first compare two source revision systems. Section 4.4.2 describes the conceptual difference between Jazz and CVS. Section 4.4.2 and Section 4.4.2 describes their implementation details respectively.

Jazz vs. CVS

The biggest difference between CVS and Jazz is that Jazz pushes the idea of local workspace one step further: Jazz stores local workspace on the server as repository workspace while there is no equivalent for CVS. A repository workspace is the personal workspace with project files that are under source control. In Jazz, teams use a stream to store the master copy of project
files; each repository workspace holds a copy. Change sets checked in to a repository workspace can be delivered to a stream to make them available to other team members.

There are no branches in Jazz since each team member is already working on a branch (called a repository workspace) from the stream and contributes to the stream in the unit of change sets. A stream is a repository workspace owned by all team members, whereas a repository workspace is a workspace owned by a particular team member. A new repository workspace can be cloned from any of these, which is similar to creating a new branch from head/branch using equivalent terms in CVS.

Another difference is that CVS does not have the concept of change sets since CVS stores revisions of files individually. In Jazz, a change set is a grouping of related changes to files in a workspace or stream. In addition to updates to the contents of files, a change set in Jazz also tracks file and folder creations, deletions, renames, and moves.

Table 4.2 provides a comparison of terms between the two version control systems.

Table 4.2: Comparison of terms between Jazz Team Server and Concurrent Version System

<table>
<thead>
<tr>
<th>Concurrent Version System</th>
<th>Jazz Team Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>Stream</td>
</tr>
<tr>
<td>Branch</td>
<td>Repository workspace</td>
</tr>
<tr>
<td>Top level folder in a module</td>
<td>Component</td>
</tr>
<tr>
<td>Commit</td>
<td>Deliver</td>
</tr>
<tr>
<td>Update</td>
<td>Accept</td>
</tr>
<tr>
<td>N/A</td>
<td>Change set</td>
</tr>
</tbody>
</table>

In the framework, we use the following terms. A delivery means committing changes to the stream from any workspace. The workspace can be a local workspace or a repository workspace. An acceptance means accepting changes from the stream to any workspace.

**Jazz Team Server**

To implement the Jazz Team Server monitor, we use the unified APIs provided by Rational Team Concert. Rational Team Concert is a client to the Jazz Team Server built on top of the Eclipse IDE and it has a set of APIs to track delivery and acceptance of change sets inside the IDE. The APIs provide access to detailed information about a change set, including the author,
4.4. Monitor

the time stamp of the change, and the change type. Individual commands are mapped, per artifact, onto an authorship event as follows:

- commit, deliver and check-in: DELIVERY
- accept and load: ACCEPTANCE

Figure 4.7 shows the dependency of the Jazz Team Server monitor.
Figure 4.7: Jazz Team Server monitor plug-in dependency
Concurrent Version System

CVS does not have the concept of a change set, hence no APIs are provided to access information concerning to a change set such as who makes the change, when was the change accepted by a particular team member and so on. In CVS, change information retrieved from the repository does not include details about the acceptance date of the changes. Thus we make an assumption that whenever a developer commits a change to the repository, other team members accept it immediately. The particular mapping of commands to events, however, is slightly different (once again, per artifact):

- *edit, commit and check-in: DELIVERY*
- *update and check out: ACCEPTANCE*

Figure 4.8 show the dependency of the CVS monitor. To implement the CVS monitor, we use the APIs that come with the Eclipse CVS client. The CVS monitor plug-in depends on the monitor framework, the Eclipse team plug-in (org.eclipse.team.core), and the Eclipse CVS client plug-in (org.eclipse.team.cvs.core).
Figure 4.8: Concurrent Versions System monitor plug-in dependency
4.5 Extensions

The framework is extensible through extension points and APIs. We define two extension points: the monitor extension point and the knowledge model extension point. With the monitor extension point, additional monitor implementations can be integrated into the framework to contribute to the DOK calculation lifecycle. With the knowledge model extension point, the framework can be extended to support new knowledge indicators, for example, adding an authorship indicator that considers information from bug reports.

Section 4.5.1 describes extending the monitor to capture authorship changes for different source revision systems. Section 4.5.2 discusses extending the knowledge model with additional indicators.

4.5.1 Bridging to a Source Revision System

The monitor component (ca.ubc.spl.cortana.monitor) provides an extension point (ca.ubc.cs.spl.cortana.monitor.repository) to monitor additional source revision systems for the DOA calculation. Figure 4.9 shows the class diagram of the repository bridge. AbstractRepository models the basic behaviors of a repository. AbstractRepositoryBridge provides operations specific to a repository such as issuing an indicator event when authorship of a source code element is changed (IIndicatorEvent). To monitor additional repositories, one must provide specific implementations of AbstractRepository and AbstractRepositoryBridge. Currently two repositories are bridged: Jazz Team Server and Concurrent Versions System.

4.5.2 Adding a Knowledge Indicator

The core plug-in (ca.ubc.cs.spl.cortana.core) provides an extension point (ca.ubc.cs.spl.cortana.core.indicator) to add new knowledge indicators into the DOK calculation lifecycle. As shown in Figure 4.2, one needs to simply provide an implementation of IDegreeOfKnowledgeComputationStrategy and IDegreeOfIndicator, which are used by IDegreeOfKnowledge to compute DOK values. Currently two indicators are implemented: degree-of-interest (IDegreeOfInterest) and degree-of-authorship (IDegreeOfAuthorship). The default computation strategy is implemented according the formula described in Section 3.3.3. This flexibility enables easy integration of additional knowledge indicators, for example, adding another authorship indicator by crawling information from bug reports.
4.6 Performance

Our event-based approach scales even if there are multiple monitors issuing thousands of events at the same time. We describe two performance test cases using the Eclipse performance test plug-in\(^6\) to demonstrate the scalability of our approach. The performance test plug-in provides infrastructure for instrumenting programs to collect performance data for Eclipse-based applications and to assert that performance does not drop below a baseline.

The goal of the performance test case is to simulate a typical workday scenario of a developer and then measure the performance of the framework's event-based model. We generate test data by assuming that the probability that a developer performs operations (e.g., a selection or a delivery) on a certain source code element conforms to a Normal Distribution, and the probability of the type of operations performed conforms to a Uniform Distribution. The reason is that typically developers focus on a subset of source code elements in a workday and the probability of operating on these elements is much higher than the rest, while the type of the operations

\(^6\)http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.test.performance/, verified 26 April 2009
4.6. Performance

performed can be regarded as random. In both our scenarios, we assume that there are 100,000 elements loaded in a developer's workspace and a developer frequently works on 1% of them (1000). We generate indicator events for these 1000 random elements with high probability and generate events for elements outside this set with low probability. The event type of the generated event is randomly picked. All the test results are statistically significant with 6400 samples and generated on a machine with a dual core 2.5GHz CPU and 3GB of RAM.

4.6.1 One Working Day Scenario

From the data collected in our previous experiment [7], a professional developer typically has an average of 7861(±3982) interaction events over seven working days and an average of 153,240(±46,572) authorship events over three months. Thus the number of events in a regular working day is approximately 2825(±1086). We conducted four performance tests to see whether the performance of the event-based model drops significantly when handling 1500, 3000, 4000, 6000 and 8000 events at once for the calculation of DOK values. The numbers of the events in three of the tests match the possible numbers of events that a developer may have in a typical workday (1500, 3000 and 4000) and we added two load tests (6000 and 8000) to investigate whether the event-based model scales.

Figure 4.10 shows the amount of Java heap used for the tests, which is the amount of memory (KB) used in the JVM calculated by subtracting free memory from total memory. The framework only needs 279.92KB memory with 8000 events contributing to the calculation of the DOK values. The amount of memory used does not double even if the number of events double from 4000 to 8000. Figure 4.11 shows the CPU time used for the same tests. The framework only needs 17ms to handle 8000 events in the load test. We conclude that there is no significant performance impact when computing DOKs in a single session.

4.6.2 Seven Working Days Scenario

We conducted another performance test to simulate a developer working consequently for seven workdays. For each day, we assume that the developer generates 8000 events (e.g., a SELECTION event) and the application will reproduce the elements with positive DOK values the next day. This simulation is close to a developer working in an IDE with one session per day. Table 4.3 shows the number of elements with positive DOK values at

35
4.6. Performance

Figure 4.10: Used Java heap in kilobytes for one simulated workday

Figure 4.11: CPU time in milliseconds for one simulated workday
4.6. Performance

Table 4.3: Number of elements with positive DOK values

<table>
<thead>
<tr>
<th>Day 1</th>
<th>Day 2</th>
<th>Day 3</th>
<th>Day 4</th>
<th>Day 5</th>
<th>Day 6</th>
<th>Day 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>767</td>
<td>780</td>
<td>798</td>
<td>779</td>
<td>811</td>
<td>782</td>
<td>772</td>
</tr>
</tbody>
</table>

the end of each day. Internally, the framework reproduces elements by issuing indicator events with the INITIAL_XXX event type (details in Table 3.2 and Table 3.4). Our test case mocks up this mechanism by translating elements with positive DOK values on the first day to indicator events before inputting them with other randomly generated events to the next day.

Figure 4.12 shows that the event-based model uses a steady amount of memory (286.5 KB on average) to handle events each day for seven working days. Figure 4.13 indicates that the event-based model needs an average of 19 ms to handle the same amount of events each day. We conclude that our event-based model scales between sessions.
4.6. Performance

Figure 4.12: Used Java heap in kilobytes for seven simulated workdays

Figure 4.13: CPU time in milliseconds for seven simulated workdays
Chapter 5

Discussion

In this chapter, we discuss the potential usefulness of the DOK model. We report on five usage scenarios that consider the use of the knowledge indicator to help scope structured views (Section 5.1), to help raise awareness of development process (Section 5.2), to help find experts (Section 5.3), to help with onboarding (Section 5.5), and to help with assessing information in change sets (Section 5.4). We also provide direction for future work (Section 5.6).

5.1 Scoped Structured View

The DOK model can help focus structured views by projecting the source code elements with high DOK values onto a particular UI facility. The basic idea is if a source code element’s DOK value exceeds a certain threshold, it is filtered in the structured views. We believe this filtering can help relieve developers from overloaded information and focus their attentions on the elements that potentially need further review.

We illustrate how to use our DOK model to further filter elements in a task context in Mylyn. As a developer works on a task, she gradually forms the task context, the set of all artifacts related to the task [9]. Mylyn uses this task context to focus the UI on interesting information, hide what’s uninteresting, for example, filtering elements in the Package Explorer view in Eclipse. With our knowledge model, we can further filter the elements in a task context that the developer already knows and focus the view on elements that might need special attention, for example, focusing on significant elements related to the task for which the developer has no knowledge. We believe this usage is helpful in scoping elements in a task context, especially if the task context is not authored by the developer but shared by other developers in the team and is retrieved from a bug repository.

Figure 5.1 illustrates this scenario with the elements in a task context on the left and elements after being filtered by the DOK model on the right. As shown, a developer uses a task context originally created by a teammate. The Mylyn focus UI is turned on (1 in Figure 5.1) and only the interesting
5.2 Awareness of Knowledge

The DOK model can be used to raise the awareness of potential increases and decreases in a developer's knowledge over a source codebase. The basic idea is that whenever a certain number of source code elements with their DOK values rises above, or drops below, a certain threshold, the developer is notified that her knowledge of these elements potentially change significantly and she may need to review them to understand the cause.

For example as shown in Figure 5.2, whenever a developer synchronizes her workspace with the source repository and plans to accept changes from her teammates, an assessment is conducted to indicate the effect on accepting these changes and a notification dialog is shown up to make the developer aware of the effect. The notification provides information such
5.2. Awareness of Knowledge

as which source code element will have its DOK value drop if the changes are accepted and the significance of the drop (e.g., 14 out of 22 elements in IKnowledgeContext will have their DOK values drop and the overall drop reaches 70%). The developer can then review these elements to understand the cause of the knowledge drop by opening the Knowledge Explorer view (e.g., the drop is caused by accepting a change from another developer in the team).
Figure 5.2: Awareness of knowledge drop when synchronizing workspace
5.3 Knowledge Map

By ranking the results of individual developer’s knowledge in a codebase using DOK, we can apply our approach to identify which team member is the expert of each part of the code base. The basic idea, as described in our user study [7], is that for each source code element in a package, and for each developers in the team, we calculate the aggregate DOK values for a package by summing the DOK values of all source code elements belonging to the package. We can then produce a diagram called knowledge map that shows for each package, the developer with the highest DOK for that package. A part of the knowledge map for a project is presented in Figure 5.3. It shows which team member knows most about which package by highlighting them with different colors. We reported a 25% improvement in correctly finding experts compared to other approaches [7].

5.4 Change Set Assessment

A developer who receives a change set from another team member typically assess the change set before accepting it into her development environment. We believe shipping with a change set, additional elements that represent the
5.5. Onboarding

The basic idea is that whenever a developer delivers a change set, she attaches to the change set the changes' structurally related elements (e.g., superclasses) which also have high DOK values, and then team members can better assess the changes by looking at these attached elements. We found that this approach shows promises in helping developers understand a change set [7].

5.5 Onboarding

We believe our DOK model is also useful in helping developers learn the basic structure of the codebase who newly join a development project. The process of becoming proficient with a codebase is known as onboarding [4]. The idea is filtering the display of code elements in a project to focus on those with which the newcomer's mentor has familiarity by ranking the elements according to their DOK values; the higher the DOK values of the elements in the mentor's workspace, the higher priority that the newcomer should learn them. We reported that the elements with high DOK values from the mentor's workspace can be used as a starting point for the newcomer to find useful elements that help learn the structure of the codebase [7].

5.6 Future Work

We have defined a model that indicates the knowledge of a developer and implemented an effective framework for calculating the DOK values. Our implementation is one particular point in the design space of using the DOK model to scope down the information presented in an application. In this section, we discuss alternative approaches and future work that could improve accuracy of the model and applicability.

5.6.1 Adding More Indicators

To gauge the knowledge of a developer, we only take into account two main factors (interactions and authorship) from [6]. To gain a better indicator of program structure knowledge, other factors such as code patterns can be used to better gauge the knowledge if proven by future experiments. As described in Section 4.5.2, the current implementation is flexible enough to support additional indicators integrated to the life cycle of the DOK computation. What is left to do is conducting user studies to find out how
significant other factors are in gauging developers’ knowledge and provide corresponding implementations.

5.6.2 An Automated Learning Model

It would be interesting to combine the DOK idea with a machine learning approach. Currently, we predefined a fixed model and apply it to developers working in various environments. While the model approximates a developer’s knowledge reasonably well, there are cases where it does not hold because of different development processes or different projects. For example, we reported that the DOK values computed using the model tuned at one site is less statistically significant at a different site [6]. We believe that each developer should have his own DOK model with a unique formula. The basic idea is using an online learning approach to keep tuning the DOK model as a developer works; by inputting with a generic model, a developer’s DOK model is constantly tweaked according to his working habit as more activity data is put in through monitors. The model will finally grow to a shape that predicts a particular developer’s knowledge specific to a particular project that he works on and a particular development process that he uses.
Chapter 6

Summary

The integrated development environments (IDEs) used by developers to work on large software systems make it easy to browse and query the structure of the information that developers need to access. However, given the complexity of today’s information systems, developers end up spending an inordinate amount of time filtering the information by considering both implicit and explicit cues. In this thesis, we investigated an indicator of a developer’s knowledge of code that might be used as an explicit cue to help developers filter through the information comprising a system.

The thesis introduces a degree-of-knowledge (DOK) model that incorporates a long-term component degree-of-authorship (DOA) and a short-term component degree-of-interest (DOI), to help combat the information overload problem faced by software developers. The thesis also describes the implementation of the DOK model in detail, provides documentation on some of the APIs and provides performance test data to show that the implementation scales. Finally, the thesis discusses how the DOK model might be investigated in future research to help alleviate the information overload problem.
Bibliography


[8] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stephane Ducasse. How developers drive software evolution. *Proceedings of the 8th Inter-
Chapter 6. Bibliography


