Can Development Work Describe Itself?

Walid Maalej
Technische Universität München
Munich, Germany
maalejw@cs.tum.edu

Hans-Jörg Happel
FZI Forschungszentrum Informatik
Karlsruhe, Germany
happel@fzi.de

***Abstract*—Work descriptions are informal notes taken by developers to summarize work achieved in a particular session. Existing studies indicate that maintaining them is a distracting task, which costs a developer more than 30 min. a day. The goal of this research is to analyze the purposes of work descriptions, and find out if automated tools can assist developers in efficiently creating them. For this, we mine a large dataset of heterogeneous work descriptions from open source and commercial projects. We analyze the semantics of these documents and identify common information entities and granularity levels. Information on performed actions, concerned artifacts, references and new work, shows the work management purpose of work descriptions. Information on problems, rationale and experience shows their knowledge sharing purpose. We discuss how work description information, in particular information used for work management, can be generated by observing developers' interactions. Our findings have many implications for next generation software engineering tools.**

I. INTRODUCTION

Developers informally describe their work in personal notes, time diaries, source code comments and commit messages. Personal notes reflect the work and capture experience, which might be useful at later occasions [8], [20]. Time diaries capture developers' activities and the time spent, such that effort can be assigned to appropriate projects [7], [11]. Source code comments include annotations such as TODO and FIXME to support articulation work [17]. Commit messages summarize the changes and convey rationale [14].

Work descriptions can be useful when communicating work status to collaborators or when resuming postponed work. They help to remember encountered issues and important decisions [15]. This paper is part of larger research that aims to systematically analyzing the nature of work descriptions and defining a conceptual and tool framework to support their creation and usage. Previously, we reported on an exploratory study [13], where we found that maintaining work descriptions costs each developer in average 30 minutes a day. About 10% of the records are meaningless (we call them pseudo descriptions). We found that developers do not find time, are not motivated or are not able to summarize these sessions. We also identified several regularities between analyzed work descriptions. The manner of using the vocabulary seems to be predictable and the vocabulary size is small, as reported by other studies [1].

These regularities motivate the main research question of this follow-on paper: To which extent can work descriptions be automated? That is, can development work describe itself? The contribution of this paper is threefold. First, it identifies information entities and information granularity included in work descriptions. Second, it defines a common model and a set of heuristics that can be used to predict work descriptions based on developers' observation. Third, it provides valuable insights into how the purposes of work descriptions can be systematically achieved – something the next generation tools should take into account.

We first introduce the used data and research method (section 2). We then describe the results of our analysis of information entities, information granularity and developers' preferences (section 3). Afterwards we present our findings on if and how work description information can be automatically created (section 4), and discuss the consensus and dissent with related work (section 5). Finally we give insights on the implications and limitations (section 6) as well as the conclusions (section 7) of our research.

II. RESEARCH SETTING

We define work descriptions as *informal text notes taken by knowledge workers to summarize achieved work and other notable issues in a particular session*. As opposed to planing artifacts that have a prospective view on the work (e.g., tasks or action items), work descriptions have a retrospective view. That is, planing artifacts include information on what should be done, whereas work descriptions include information on what has been done. Our research focuses on the content of work descriptions, given the start and end of a work session. We assume that the developer's workday is already split in different sessions. Thus, the goal is not to analyze when or how developers change their focus to work on a different task, but to analyze descriptions of *given* sessions and investigate means of automating these descriptions. Hereafter, we describe the data used in our research and the overall research method.

A. Work Description Data

Our data is selected to reflect different types of work descriptions and projects. The first dataset (MYCOMP) consists of 70,556 personal work logs and time cards of the

38 MYCOMP employees. MYCOMP is a German medium-sized software company offering Internet-based services. Its employees continuously logged their work sessions in a database and charged their work time (session durations) to particular projects. The data is in German language and was collected throughout a period of eight years between 2001 and 2009. The second dataset (APACHE) consists of 643,252 commit messages and code comments submitted to the central version control system of the APACHE SOFTWARE FOUNDATION[1]. They describe work performed in one of the 73 APACHE projects between 2001 and 2009 by 1,949 committers. APACHE projects typically include complex infrastructure modules, have high coding standards and involve a large number of globally distributed developers. Although the foundation uses a central code repository, each project has its own standards and processes. The third dataset (UNICASE) consists of 5,435 commit messages and code comments of the open source project UNICASE[2]. The texts are in English and describe work performed in 2008 and 2009 by 37 developers. Unicase is a CASE tool integrating different software engineering artifacts, such as requirements, use cases, UML models, schedules and bugs into a unified model. As opposed to APACHE, UNICASE is a single project carried out by collocated developers and graduate students from the Technische Universität München. The fourth dataset (EUREKA) was gathered in an observational study conducted with 21 developers from five different European software companies. The goal of the study was to collect machine-readable interaction data and human-readable work descriptions from professional developers. The dataset consists of 115 reports in English. MYCOMP, APACHE and EUREKA were used in our previous study [13].

To prepare the data for the analysis, we removed all pseudo descriptions (about 10% of the entries [13]). Pseudo descriptions are meaningless. They consist of empty strings, special characters such as "-" or "00", or too short strings (less that 2 words and 8 characters, "fix bugs" is a meaningful description). A developer must have at least 50 descriptions within a period of 2 months to qualify for the analysis. Source code included in the descriptions is removed and substituted with a placeholder. Some developers use special characters to visually format the text, e.g. draw a tabulator with "++++". We ignored these parts because a) we consider them to be structured and b) they cannot be parsed by natural language processing tools. We used regular expressions to filter code commented out, automatically inserted annotations, JavaDocs and license text from the source code comments. We also removed bunches of comments included at once, typically added in code clean-ups. Comments that belong to one commit are considered as a single work description. Table I shows the prepared data.

[1]http://svn.apache.org/repos/asf/

[2]http://unicase.googlecode.com/svn/trunk/

Table I
DATASETS USED IN THE ANALYSIS.

Dataset	Type	Period	Entries	Developers
MYCOMP	personal notes & time sheets	2001 - 2009	38,005	25
APACHE	commit messages & code comments	2001 - 2009	598,418	1,145
UNICASE	commit messages & code comments	2008 - 2009	5097	18
EUREKA	Logs of an observational study	2008	91	21

B. Research Method

The central question that drives our research is: to which extent can work descriptions be automated? To answer this question, we study a large number of work descriptions and empirically identify regularities in their contents. We analyze the *information* included in the work descriptions as well as the *behavior of developers* in describing their work. Analyzing the information involves identifying frequent *information entities* (text fragments with similar semantics). This is analogous to creating structured forms where we can integrate the unstructured work description texts. Moreover, we analyze the different levels of detail included in work description information. We call this *semantic granularity*, which refers to abstraction levels of included information. We aim at identifying various granularity levels and identifying possible correlations with other properties.

Analyzing developers involves looking for clusters based on their describing behaviors. That is we examine if particular developers typically include particular information with a particular granularity level. By comparing work descriptions gathered in different tools we aim at identifying possible commonalities and differences and how developers use these tools to capture information on their work.

For analyzing work description information we use techniques from natural language processing (NLP) [2]. First, in a theory building phase we systematically identify candidates for information entities and granularity levels. We tokenize the unstructured text into word categories of the English and German languages. Here we use the largest datasets APACHE (English) and MYCOMP (German). We study the categories: noun phrases (e.g. "interface specification"), verb phrases (e.g. "kept testing"), adjectives, adverbs (e.g. open, not), cardinals, conjunctions, propositions (because of) and symbols. By identifying top frequent words in each category and comparing their ranks in common languages, we aim at hypothetically defining recurrent information. In NLP, part-of-speech (POS) tagging automatically assigns POS tags (nouns, verbs etc.) to each word in work description text, while accounting for words that have different meaning but identical spellings, like "a release" and "to release". For example given the sentence "The bug is serious", the output would be "The/Article bug/Noun is/Verb

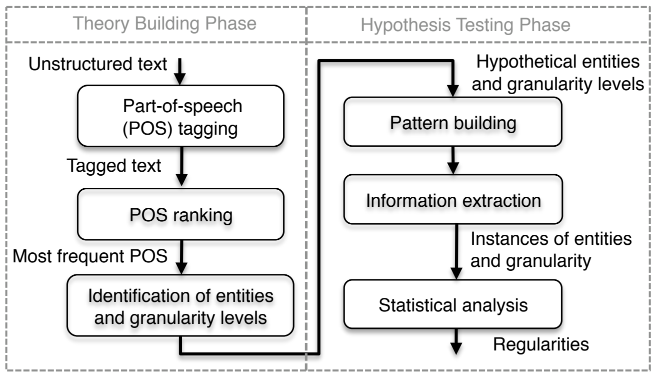

Figure 1. The data analysis process.

serious/Adjective". We use the Stanford Maximum Entropy POS Tagger which has, equally for English and German text, an accuracy as high as 97% for known and 90% for unknown words [19].

Then, in a hypothesis testing phase we quantify our observations. For that we apply information extraction techniques [2] on all datasets, to systematically extract instances of identified information entities from each work description. These techniques base on pattern matching, predefined instance lists and supervised classifiers. We use the Stanford Named Entity Recognizer (NER), trained with the annotated corpora CoNLL-2002 corpus [18] and the Problem Report Corpus [10]. We build a set of patterns (regular expressions of tags, keywords and synonyms using Concise Oxford English Dictionary and Thesaurus[3] and the open thesaurus[4]). As a result, work descriptions are annotated with frequencies of each information entity and granularity level, developer's name and session duration. This allows for performing statistical queries to identify possible relationships and correlations. Figure 1 summarizes the data analysis process.

III. Work Description Analysis

We first report on results for the part-of-speech tagging, and identify candidates for commonly used information entities. Then we describe the extraction patterns and report on the frequencies of information entities and granularity. Finally we summarize our analysis of developers' behavior.

A. Parts of Speech

Verb Phrases: There are 22,074 distinct English and 3,353 German verbs in Apache resp. Mycomp dataset. Only 804 (3%) distinct English and 1164 (34%) German verbs constitute 90% of all verb occurrences. German verbs are often combined with prefixes and have many variations (e.g. fügen, einfügen, hinzufügen, zusamenfügen), which explains this high percentage of distinct German verbs compared to English ones. Of the 804, respectively 1164

[3]http://www.babylon.com/dictionary/oxford.html
[4]http://www.openthesaurus.de/about/api

Table II
Most frequent verbs in the datasets.

Verb	# Apache	% verbs	# Mycomp	% verbs
add	121,141	10.1	1,591	6.2
fix	67,796	5.7	680	2.6
remove	44,409	3.7	559	2.2
change	13,317	1.1	763	3.0
create	11,539	1.0	2,522	9.9
test	8,910	0.7	500	2.0
read	2,122	0.2	531	2.1
meet / review	136 / 15,842	0 / 1.3	2,767 / 184	10.8 / 0.7

verbs, there exist 305 English and 288 German base forms. Table II shows most common verbs, their frequencies and their proportion of all verb instances. From the top 1000 verb phrases we filtered those with >2x higher ranking than in natural language texts. This results in 418 verb phrases. We manually analyzed these verb phrases and found two categories of meaning. The first category is the larger one and includes verb phrases such as "add, fix, removed, merge, test or debug". These verbs denote actions and activities performed by the developers in the work sessions. The second category includes verb phrases such as "to do, need to, should be, have to, missing", which denote work that is open, need to be competed or that should be done. Examples of such work descriptions are: "We need to refactor the fixtures of tests", "Selection still needs to be passed to the action". We hypothesize that *performed actions* as well as *identified work* are two information entities commonly included in work descriptions.

The verbs in the performed actions category have three different semantics: edit, process and knowledge related action. *Edit* related verbs are document-editing verbs, such as "remove, edit, add, delete". *Process* related verbs refer to software engineering specific activities such as "test, debug, specify, implement". *Knowledge* related verbs consist of knowledge acquisition and sharing verb phrases such as "learn, try, solve, find, seek, checked". Examples are "Tried to load different analyzer configurations from analyzer wizard", "learned how to use PHP-Nuke". While verbs related to edit actions are by far the most highly ranked in the English work descriptions, in the German work descriptions process-related verbs have similar frequencies to edit verbs.

Noun Phrases: There exist 234,202 distinct English and 13,558 German nouns in Apache resp. Mycomp dataset. Table III shows most frequent ones. By manually analyzing the top 1000 nouns we found in both languages about 700 nouns that represent *used artifacts*, i.e. physical or logical entities affected by developer's actions, e.g. the edited method, used library, read documentation, or conducted test. These entities make out at least 37% of all nouns in the German respectively 45.9% in the English data. Top ranked English nouns are code-related artifacts: "file", "class", "method", "version" and "package", while

Table III
MOST FREQUENT NOUNS IN THE DATASETS.

Verb	# APACHE	% nouns	# MYCOMP	% nouns
test	61630	2.4	1557	1.9
file	42465	1.7	732	0.9
fix	32202	1.3	550	0.7
bug	31713	1.3	1045	1.3
code	24506	1.0	254	0.3
version	20844	0.8	186	0.2
release	16538	0.7	614	0.8
problem	10183	0.4	982	1.2

top ranked German nouns are “emails”, “page”, “image”, “file”. Example work description including this information are: “Create TestLoadXMLTypeFromOracle class to the test found in the Internet for Oracle XmlType”, “I had to remove workspace/.metadata/.lock”. The second group of highly ranked nouns represents *problems* or difficulties encountered during the development work as well as followed *solutions* or solution concepts. For example “Fixed table scroll problem...”. The Nouns “bug, problem, exception, issue, solution, fix” count about 10% of all nouns in both languages. We hypothesize that artifacts used, encountered problems as well as used solutions are information entities commonly included in work descriptions.

The nouns representing artifacts reflect three abstraction layers. In the English data about 31% of these nouns refer to *implementation entities:* objects from the solution domain such as designed interfaces (e.g. “LoadXMLType”), used databases (e.g. “Oracle”) and libraries (e.g. “eversuite framework”). About 8% refer to *requirements entities:* objects from the problem domains and requirements of systems under development. Examples for such information are “the drag and drop feature”, “new security requirement”, “client X change request”. Finally about *6%* refer to *project entities*, planing and project management information, such as “1.0.2 release, task #2208”. Nouns representing implementation artifacts represent four abstraction layers from the object-oriented programming paradigm. Many noun phrases refer to *components* such as “the separate plugin com.mycomp.utils.sortableTable”. Other nouns refer to *classes* and objects such as “IResourceDelta objects” or “manager class”. Others refer to *methods* such as “isDirty() method” or even more fine-grained pieces of code such as “a flag to indicate...”, “the if-statement”.

Adverbs and Adjectives: We found 22.2% of the work descriptions include adverbs while 38,92% include adjectives. In the English dataset (APACHE) 2,636 distinct adverbs and 44,373 distinct adjectives. In the German dataset (MYCOMP) we found 758 distinct adverbs and 5,070 distinct adjectives. Top ranked adverbs are “not, now, instead, only, just, still, correctly, properly, still”. While “not” is also top ranked in both German and English general-purpose texts, we observe unexpectedly high usage of *status* adverbs. About 30% of all adverbs used in MYCOMP (German) and 48% of the adverbs used APACHE (English) denote status. Example texts are “Building the Xalan2 from the latest sources works fine now” or “Dynamic loading of articles works now”. Similarly, most frequently used adjectives are “new, done, correct, possible, better”. We hypothesize that *status* of work is a common information entity in work descriptions.

Conjunctions and Prepositions: Conjunctions and prepositions (C&P) serve two roles in the analyzed datasets. The first role is to contextualize developers’ statements. The terms “if, by, while, during, when, in, of, at...” are among the top-40 list of all 1000 distinct C&Ps. These terms are used in 68.2% of the work descriptions and constitute more than 60.3% of all C&P instances. These terms denote context of particular “truth”, knowledge and experience, developers have “learned” in the work session. Examples of such descriptions are “if you want to check the result of dialogs in swt... you can use...” or “when a pool tunneled is required...”. The second role of conjunctions and prepositions in the datasets is to give justifications for a particular action, change or a decision. The terms “for, since, because, as, so” are used in 28,2% of the work descriptions and constitute 23.9% of all used C&Ps. Their ranks in the datasets are at least 10x higher than in English and German common language texts. By using these terms developers answer questions on why decisions have been taken or changes have been done in a particular way. An example of a work description is “Updated the status... *since* we only pull the server date every minute.” We hypothesize that developers commonly include *rationale* and *experience* information in their work descriptions.

Numeral, Cardinal and Symbols: Our datasets include 137,343 numerals such as “.2, one, 9813, 0.25 ”. We found that numerals play two roles. First, 24% of the numerals are combined with the symbols and acronyms “#, PR, no, BUG, CR, id, TASK, AP” (AP is German abbreviation of work package). These numerals present *references* to planing artifacts such as task description documents, bug reports and change requests. Second, 37.1% of the numerals are combined with nouns such as “version, v, revision, release” as well as names of systems like “MySQL, Oracle, Java”. This supports our previous observation about information on concerned artifacts. Concerning the use of symbols, “:” and “-” are by far highest ranked ones and are included in about 20.3% of both English and German work descriptions. These symbols separate brief descriptions and a more detailed explanations and rationales. The other frequently used symbols are “//” and “/” which are combined with acronyms such as “http”, “C”, “file”, “bin”. Such information also represents references to artifacts concerned by the change or artifact containing more information on change and work backgrounds. We hypothesize that *references* to planing artifacts, concerned artifacts and cause artifacts are commonly included in the work descriptions.

Table IV
INDICATORS OF INFORMATION ENTITIES.

Information	Indicator
Action	Identified activity verbs [13], their synonyms in the base or past form, usually conjugated with first person singular.
Artifact	Artifact types keywords "method, class, component, interface, file..." with a named entity. Patterns of file names (e.g. *.java). Patterns of identifiers.
Reference	Patterns for protocols e.g. "http://". Keywords "see, more info, refer to", abbreviations with cardinals.
Status	Status keywords "done, open, closed, worked,..." combined with artifact names "task, work, bug, issue..."
New Work	Keywords "todo, fix me, revisit". "need to, should, must, have to" combined with <Action> and <Artifact>
Problem	Keywords "problem, issue, bug, failure, exception, block, need". Exception names and abbreviations
Solution	Keywords "solution, fix <verb>, solve, resolve, " combined with <Problem>
Rationale	Keywords "for, because, since, in order to, reason, rationale, justification" combined with <Action>
Experience	Context keywords "if, when, by..." with imperative phrase. Keyword "learned, found, recognized"

B. Information Entities

Our observations resulted in 9 hypothetical information entities that are commonly included in work descriptions. To check these observations we determine the frequency of each entity by extracting its instances. Table IV shows major indicators and patterns we use in the information extraction. Table V summarizes the results which confirm our observations. Actions performed in the work session represent the most frequent information entity, included in 70.6% of the descriptions on average. Experience is the less frequent information, included on average in every tenth description. MYCOMP includes below-average experience information, which can be a consequence of the asynchronous capture of the work descriptions, typically by the end of the work day [13]. In 4% of work descriptions we were not able to extract instances for any of the identified entities. We randomly sampled 300 of these work description to manually investigate them. We were not able to identify common information ($CI = 5\%, p = .05$). About half of these entries included non-meaningful information such as "A long day", "just comment", "** empty commit message**". The other half includes information as diverse as source code, pseudo code, discussion extracts or timing information.

By analyzing the co-occurrence of information entities, we found several interesting trends. 82% of information on performed actions is combined with concerned artifacts. The "information template" *action concerns artifact* is thus widely used by developers. About 18% of the actions do not affect any particular artifacts. This can result from errors in the information extraction algorithm. This might also result from knowledge acquisition and sharing actions such as learning a technology, searching a keyword, reading. In MYCOMP we found many of these verbs to be "do nothing, have break". Information on *status* and *identified work* is also combined frequently – describing work done and related work that still needs to be done. Problem information is combined with actions (20.6%), experience (18.1%), status (25.3%) and identified work (14.1%). Problems seem to be "part of the work done", i.e. solved problems, which can be different from discovered problems (to be solved). Problems are also used to describe experiences, i.e. the context of acquired knowledge. References are combined with status (12.0%), rationale (24.0%), and experience (30.1%). The combination patterns shows that sharing knowledge and managing work and collaboration are two concerns of developers while annotating their work.

Table V
WORK DESCRIPTION INFORMATION.

Information	Occurrences %				
	APACHE	MYCOMP	UNICASE	EUREKA	Avg.
Action	69.3	75.9	70.5	66.6	70.6
Artifact	60.2	53.4	49.1	58	55.2
Reference	15	18.9	17.2	10.1	15.3
Status	23.5	20.3	17	15.2	19.0
New Work	24.4	19.7	28.3	21.7	23.5
Problem	46.8	47.1	48.9	45	47.0
Solution	18.7	15	15.6	10.9	15.1
Rationale	30.2	28.6	24.6	30.5	28.5
Experience	11.1	13.2	9.4	6	9.9

C. Information Granularity

Table VI summarizes indicators for identified levels of the activity, domain and object granularity. 62% of the work descriptions include information from one single granularity level. 12% of the work descriptions with artifact information include only requirements artifacts, 31% include only project artifacts, while 54% include only implementation artifacts. MYCOMP data included more requirements and project information. Object granularity levels are used in a disjoint way too. Only 6% of the work descriptions included information from multiple levels (e.g. method and class). It seems that developers consistently think in a single abstraction level when taking notes about the work. In contrast, levels of activity granularity are overlapped. Edit activities are combined with process activities (49%) and knowledge activities (11%). Only 13% of knowledge activities are combined with activities from other levels. Edit activities are frequently combined with implementation artifacts (70%). Process activities are combined with information on requirements and components (32%).

By correlating session durations with the granularity levels, we found interesting relationships. 87% of sessions that are shorter than 90 min. include information on methods or lines artifacts in their descriptions, while 91% of sessions that are longer than 90 min. include information on compo-

Table VI
INDICATORS FOR GRANULARITY LEVELS.

Granularity	Indicator
Requirement	Requirements keywords “feature, scenario, use case, functionality, spec.” or synonyms, combined with requirements verbs “specify, elicit, interview...”
Project	Management keywords “plan, task, release, project, sub-project, TODO, work item, bug report, change requests”
Implementation	Keywords indicating source code entities or programming concepts
Component	“Component, subsystem, plugin, library, layer...”
Class	“Class” “.java”, patterns for class names
Method	“method, function, interface”, “()”, patterns of method names
Line	Source code snippets
Edit	“Add, remove, edit, change, update, create, write, move, read, clean up”
SE Process	“fix, test, debug, implement, specify, design, plan, release, integrate, refactor, report, meet, build, adjust, install...”
Knowledge	“find, learn, try, check, solve, get problem, seek”

Table VII
WORK DESCRIPTION GRANULARITY LEVELS

Granularity	Occurrences %				
	APACHE	MYCOMP	UNICASE	EUREKA	Avg.
Requirement	10	26	6	7	12
Project	29	34	29	30	31
Implementation	58	37	62	60	54
Component	14	16	19	10	15
Class	29	25	31	32	29
Method	33	49	28	20	33
Line	17	8	17	27	17
Edit	55	41	57	60	53
SE Process	34	42	30	39	36
Knowledge	13	15	11	9	12

nents. That is, the shorter the session is the more fine-grained the described artifacts are ($p < 0.01$). 83% of short sessions are described with an edit granularity level, while 64% of long sessions are described with a process level. We were not able to identify a significant relationship between the domain granularity and session durations.

D. Developers

By plotting the descriptions over the information entities and developers, we found two developer clusters. Developers from cluster “Prob.” use problem information to describe more than 75% of their sessions. These developers extensively combine information on performed actions, new work and experiences with problem information. We found between 26% and 31% of these developers in our datasets. Cluster “Art.” seems to be more artifact oriented. These developers use artifact information in more than 75% of their work descriptions (82.4% avg.). We found between 38% and 42% of these developers in our datasets. Table VIII shows the distribution of information entities among the developers. Concerning the granularity levels, we found one cluster of developers who particularly include requirements and project information in their work descriptions (Cluster “Spec.”). These developers seem to refer to tasks and specifications more than others. We found approximately 28% of these developers in the datasets. Table VIII shows the distribution of granularity levels amongst the developers.

Table VIII
DEVELOPERS PREFERRED INFORMATION.

Information	Avg. %	Cluster Art.	Cluster Prob.
Action	70.6	74.1	62.3
Artifact	**55.2**	**82.4**	**19.8**
Reference	15.3	14.9	16.0
Status	19.0	18.4	20.2
New Work	23.5	24.7	25.5
Problem	**47.0**	**12.3**	**77.9**
Solution	17.1	15.2	19.4
Rationale	28.5	30.3	28.4
Experience	9.9	9.1	10.7

Table IX
DEVELOPERS PREFERRED GRANULARITY.

Information	Avg.	cluster Spec.
Requirement	**12**	**70**
Project	**31**	**82**
Implementation	54	15
Component	15	12
Class	29	9
Method	33	4
Line	17	16
Edit	53	55
SE Process	39	36
Knowledge	12	14

IV. FINDINGS

A number of salient themes emerged from our data analysis. These themes are categorized within work description purposes and work description automation.

A. Work Description Purposes

The semantics of the extracted information shows that developers use work descriptions for two purposes: work management and knowledge sharing.

Work Management: Developers extensively use work descriptions to capture work status, collect issues and identify work items. We were able to extract explicit status information from one fifth of studied work descriptions. Developers also implicitly capture status by describing what they have done so far. In about 70% of studied work descriptions, they summarize relevant actions they have performed and concerned artifacts. Summarizing changes helps to assess what has been done and what is missing.

Capturing work status enables developers to (i) remember it when resuming postponed tasks, (ii) communicate it to collaborators and (iii) report progress to the management [11].

Firstly, frequent interruptions in developers' work day leads to loosing work contexts. Work description help remembering this context and rebuilding the mental model. Secondly, knowing about the progress of a collective task, or relevant changes of a shared artifact help collaborators in distributed settings to efficiently resume the work and update their own mental models. Work descriptions give a focused and human readable summary of what a particular developer changed – unlike a diff that might include too much detail. Thirdly, some employers explicitly require developers to describe what they did during a work day and how much time do they spend in their different sessions. For maintenance-intensive or fixed-budget projects, organizations need to know which work has been done in a particular period of time, how much manpower did it cost to add a particular feature, or how much budget the customer still has for an invoice period.

We were also able to extract information on new work from a quarter and emerging problems from a half of analyzed work descriptions. New work items are clearly defined to-dos and can be assigned to people. This information is combined with information on what actions should be performed (e.g. remove) and what artifacts will be affected. Unlike new work that has a planing function, emerging problems rather have a design or a collaboration function. These problems need to be discussed and refined. Developers just notice that they exist and decisions need to be taken about them.

Knowledge Sharing: Much of the information extracted from the work descriptions refer to knowledge gained during the work. We found two forms of such knowledge: rationale and experiences. Rationale information is justification of design or management decisions [4]. Our analysis shows that about one third of work descriptions are used to capture rationale. A work description is a part of the session context, which makes it an intuitive place to log major decisions and why they have been taken in a particular way. By reloading a particular artifact version, developers can reason about previous decisions by reading the descriptions of the session which resulted in this version.

Work descriptions also include major work findings and what developers have learned in the session, in form of problem solution information. The problems describe a particular context in which the experience holds. The solution is either a set of actions which affects particular artifacts, a reference to a document with the details or a description of a concept. Interestingly, developers also refer to addressees in their work descriptions using "you, we, your".

B. Automating Work Descriptions

Knowing which information constitutes work descriptions, we discuss means for generating this information. We introduce a context ontology which reflects the identified semantic entities and argue how they can be generated. We then present heuristics to be used by generation approaches to guess what is a relevant context and what is not.

Context Ontology: Based on extracted verbs describing actions and types of concerned artifact and encountered problems we constructed an ontology, representing a common context model for development work. This model can be used for generating work description information. The complete ontology is available for download on the teamweaver site[5]. It consists of the interaction, artifact, problem, organization and knowledge sub-ontologies. The hierarchies reflects identified granularity levels.

Concepts included in the interaction, artifacts and problem ontologies can be instantiated automatically by instrumenting development tools. Performed *actions* such as add, change or remove can be observed, since manipulated information (e.g. code, models and plans) forms a part of structured documents. Fine-grained actions can be used as indicators for more abstract actions. Running test scripts indicates testing. Manipulating a breakpoint indicates debugging. Editing the body of a method for a long period of time indicates implementing the method, while changing its signature means specifying interfaces. Many of the concepts of the interaction ontology are used as menu labels (e.g. search, build, or create project) and can be tracked easily by instrumenting the user interface. Metadata about concerned *artifacts* can be obtained either from the underlying file systems or used tools, which maintain in most cases artifacts metadata such as name, path, modification time and mime type. The ontological concept of concerned artifacts (e.g. method, email, component) can be inferred from its mime-type (e.g. .java, .email, .jar), its name (e.g. XMLTypeTest, Meeting Minute 22 Feb), and the type of interactions performed on it (e.g. insert method, add class). More abstract information can also be collected automatically. For example, the path of the used class often includes the project name. References are either viewed documents (e.g. a documentation or a bug report), reused libraries and frameworks (e.g. imported packages or build path entries) or used tools (e.g. compiler version). Information on simple problems can be obtained from execution logs, debugger output, exceptions, warnings and test or build results.

In many other cases generating the information is a rather difficult task. Many of the problems are of conceptual natures, and can't be detected easily, such as the unexpected behavior of a program or an unsatisfied information need of the developer. Rationale and experience is rather difficult to generate as well. Decision alternatives are usually implicit in the head of developers and do not have to be investigated in the specific work session. Similarly, experience might build on previous observations. A major difficulty to generate work descriptions is to filter which actions, concerned artifacts, encountered problems and viewed and used artifacts

[5]http://www.teamweaver.org/wiki/index.php/ontology

should be included and which not. Though our results can be used to summarize information, that can be assessed and agreed by the developer. Generated information will be adopted by developers in many cases as it is. In the other cases it sill helps developers to remember major findings and provides hints, keywords which should be included.

Heuristics: Our results show that a system that generates work descriptions should be able, not only to collect relevant context information, but also to guess (i) what is relevant (ii) the appropriate details (iii) the problem situations and (iv) developer's preferences. We discuss heuristics that enables these tasks.

- Relevant vs. irrelevant: Only a subset of artifacts concerned by the interactions is included in the description. Thus, a system that generates work descriptions should be able to find out which artifacts are likely to be included. Metrics such as the accumulated duration of artifact usage over the session, recency of artifact usage (i.e. at the session end or beginning), frequency of interactions with the artifact as well as the probability of its inclusion based on previous work descriptions of the developer and her collaborators. For example if a developer works long time on the implementation of a method, then implementation details are likely to be included in the description. If the developer touches many methods of a class then the description might refer the classes or package containing these methods.
- Appropriate detail: A system that generates work descriptions should be also able to guess the appropriate level of detail. Our results show that the granularity of work description information depends on the session duration. We found that the longer the session is, the higher are referred actions and artifacts in the constructed ontology (top level entities). In this case information on requirements or component is e.g. likely to be more relevant that a method or a class.
- Problem situation: Such a system should be able to detect if a developer is e.g. "encountering a problem", "applying a solution" or "searching for a solution". The mental situations can be aggregated by understanding the semantics of the fine-grained interactions and the semantics of the artifacts as well. For example while executing a just developed piece of code, error massages means an encountered problem. Setting breakpoints and stepping through the code reveal a problem situation as well. Terms used in the web search are relevant for describing an information need problem.
- Developer preferences: Such a system should also learn from previous behavior of developers and which information they describe in which situation. Individual profiles include problem vs. artifact oriented, work habits, time and frequency of submissions. We think that *personalization* of work descriptions is an important aspect that should be further studied.

V. Related Work

As far as we can tell, no other work studied informal work logs for the purpose of automating of the description of developers work sessions. However, there exist considerable related work. Many authors studied commit messages and found similar results to ours [1], [6], [14]. Mockus and Votta [14] used word frequency techniques to classify comments of commits. They identified four types of changes: adding new functionality, repairing faults, restructuring the code to accommodate future changes, and code inspection rework (a mixture of corrective and perfective changes). Hatturi and Lanza [6] classified the work associated to the commits of nine open source projects into forward engineering (implementation of new requirements), reengineering (refactoring, redesign and other actions to enhance the quality of the code), corrective engineering (handling defects, errors and bug in the software), management (unrelated to codification, such as formatting code, cleaning up, and updating documentation). The ontology we describe in section 4 represents similar results. However our classification is fine-grained, spans other activities than implementation work, and presents a multiple-dimensional taxonomy (ontology). Our goal is not to study the nature of maintenance work but rather to generate work description information based on a common shared knowledge model. Since also fine-grained activities like remove and update are frequently used, they are part of our model. We also focus on analyzing descriptions of general work sessions of developers work and not exclusively on commit messages. Our data enables to identify further categories beyond implementation-relevant once such as modeling, meeting or learning.

Alali, Kagdi, and Maletic [1] also examined the version histories of nine open source projects to uncover trends and characteristics of how developers commit source code. The authors extracted most common words from the commit messages. Their top frequent terms reflect our identified the top frequent noun phrases (file, code, work) and verb phrases (fix, add, remove). Using techniques from natural language processing instead of term indexing brings three advantages. First we were able to consider part-of-speeches typically ignored by indexers (stop words) such as conjunctions, adverbs and numbers. From these words we identified information entities such as rationale and references. Second, these techniques enable to go into more details of the semantics, e.g. by recognizing if the word "test" refers to an action or to an artifact. Third, we extracted information entities and concepts (whose instances might include several terms) rather than unique terms. This resulted in a more detailed analysis and classification of the data.

Studies on commits also correlated the size of commits with work categories they have identified. Mockus and Votta [14] compared the distribution of each maintenance activity with the size of a change in terms of lines added and deleted, and found that adaptive and inspection changes added most lines while inspection activities deleted many more lines than other activities. Alali et al. used term combinations e.g. {fix, remove} as category labels and observed that commit size can be indicative of the types of maintenance activities being performed. Hatturi and Lanza [6] went into more details and found that the majority of tiny commits are not related to development activities. Corrective actions are the ones that generate more tiny commits. Instead of the affected artifacts (lines, files) we measure "size" based on work session durations and correlated this to the granularity levels. We found similar results, e.g. small sessions are more edit-oriented. One can use both metrics (duration and change complexity) to get better results.

Other researchers studied unstructured text of developers. Story et. al. [17] studied source code comments and found that many of them are used to identify new work items. This conforms to our findings on new work and problem information. This work complements ours since it goes into the details of one particular work description type. We included other types of comments in our analysis and studied purposes of work descriptions (session-based) rather than single source code annotations. Etzkom, Davis and Bowen [5] analyzed the English-language comments in object-oriented software. They found that these comments meet the sublanguage criteria such as lexical, syntactic and semantic restrictions or the use of special symbols – as e.g. the 'language of biophysics' does. Our goal is to identify information patterns in work descriptions rather than identifying linguistic and pragmatic criteria. Though, many of our findings from the Part-Of-Speech analysis confirm these sublanguage criteria. Ko, Myers, and Chau [10] performed a similar linguistic analysis on how people describe software problems. Our research method overlaps with theirs, but with another focus. The findings of the problem descriptions also reflect ours in many aspects such as the concerned artifacts and problem description.

Finally, in our previous paper [13] we did not go into detail of text semantics and developer's describing behavior. The exploratory, not quantified and rather selective description patterns presented a major shortcoming of our previous research. In this paper we focus on the content aspect rather than the timing aspect. We extended the dataset to a new organization (Unicase) and different medium (source code comments). We do not aim at retrieving all possible description patterns. We aim at identifying the semantic parts for the patterns and how developers include these parts in their work session summaries.

VI. Discussion

A. Implications for Tool Vendors

Our findings have several implications for tool designers, concerning work description creation and usage. We showed *which* and *how* information included in work description can be automatically created. Providing this information as "default text" helps developers to efficiently capture their work status and gained knowledge. One can think about pre-filled commit messages, synchronous communication tools that show current focus to collaborators or automatically generated Twitter messages. The first step toward the realization of those features is to extend the tools' *context awareness*. That is, tools should be able to collect interactions and their semantics. First implementation exists already and proofed to be cost-efficient [9], [12]. It is crucial for the context awareness to use a common shared model on development knowledge (e.g. what is testing and what is a method) and to span the whole range of used tools: from the email client to the debugger. The instrumentation of the IDE is not enough since relevant information for work description can come from visited web pages, read emails or viewed models. Tool vendors have to solve at least two issues. First new tools with new interfaces, implementation and versions will continue to emerge. *Integrating* these tools to the instrumentation system is a technical challenge [11]. Second, in order for developers to accept such tools *privacy issues* must be clarified.

B. Limitations

We have made several simplifying assumptions as we proceeded with the analysis. We assumed that collected session durations in the datasets are correct [13]. It is imaginable that some developers summarize several sessions in a single description (one commit or one work log). Thus the correct durations and information sessionizations might differ from the datasets, influencing our results. By filtering very long sessions (> 8 hours) we tried to eliminate this bias. Moreover, we did not investigate when developers change their work sessions and what are indications for a new focus. Related work has proposed strategies to deal with this problem. Parnin and Rugaber [15] propose resumption strategies for interrupted programming tasks and discussed their implementation by future development tools. TaskTracer [3], [16] is a task-aware desktop environment, which recognizes task switching. Task focused programming [9] introduces one approach to organize the development artifacts around the current task. Recognizing task switches is required to automate work descriptions. For this paper we focused on the content aspect. The information extraction techniques and patterns were selected carefully and tested on 350 randomly selected work description entries. This gives us 95% confidence that our results can only vary ±5% from the real values.

We did not focus on comparing the types of work description artifacts (i.e. commit messages, comments, personal notes...). Developers may describe their work differently in different types of artifacts. So it might be interesting to see how the use of language and how the semantics of their descriptions changes according to the artifact type. For such analysis multiple datasets for each artifact type are needed.

Finally, we think that our method is appropriate to investigate the automation question. Though, we think that the final answer to this question should come from the developers themselves. The quality of generated work descriptions must be evaluated in laboratory and real world setting with real developers. Thereby it is important to study the usefulness and quality of generated work descriptions from both the information producer and consumer point of view.

VII. Conclusion

Can development work describe itself? Our research shows several regularities in the way development work is described. We found work description include one of nine information entities. Information on performed actions, concerned artifacts, new work as well as references to tasks and documentations is included to manage the collaborative and fragmented work. Information on encountered problems, developed solutions, rationale and experience is used to capture knowledge. Work management information, such as a bug report the developer reads during a work session or a document she changes, can be generated by instrumenting the work environment. From our analysis we build an ontology and a set of heuristics, which can be used to generate such information. Knowledge, in contrast, is rather difficult to generate. We think this information should be articulated by developers themselves. However tools can assist developers and remind them to capture this knowledge by understanding which problems has been solved – endorsing knowledge sharing in development teams.

References

[1] A. Alali, H. Kagdi, and J. I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*. IEEE CS, 2008.

[2] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python*. O'Reilly Media, June 2009.

[3] A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *Proceedings of the 10th international conference on Intelligent user interfaces*. ACM, 2005.

[4] A. H. Dutoit, R. McCall, I. Mistrik, and B. Paech, editors. *Rationale Management in Software Engineering*. Springer, May 2006.

[5] L. H. Etzkorn, C. G. Davis, and L. L. Bowen. The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, 33(11):1731 – 1756, 2001.

[6] L. Hattori and M. Lanza. On the nature of commits. In *ASE Workshops*, pages 63–71. IEEE, 2008.

[7] L. Hochstein, V. R. Basili, M. V. Zelkowitz, J. K. Hollingsworth, and J. Carver. Combining self-reported and automatic data to improve programming effort measurement. In *Proceedings of the 10th European software engineering conference*. ACM, 2005.

[8] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *14th FSE*. ACM Press, 2006.

[10] A. J. Ko, B. A. Myers, and D. H. Chau. A linguistic analysis of how people describe software problems. In *Proceedings of the Visual Languages and Human-Centric Computing*. IEEE Computer Society, 2006.

[11] W. Maalej. Task-first or context-first? tool integration revisited. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, Los Alamitos, 2009. IEEE Computer Society.

[12] W. Maalej and H.-J. Happel. A lightweight approach for knowledge sharing in distributed software teams. In *7th International Conference on Practical Aspects of Knowledge Management*, Lecture Notes in Computer Science. Springer, 2008.

[13] W. Maalej and H.-J. Happel. From work to word: How do software developers describe their work? In *Proceedings of the 6th IEEE Conference On Mining Software Repositories*, 2009.

[14] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2000.

[15] C. Parnin and S. Rugaber. Resumption strategies for interrupted programming tasks. In *ICPC*, pages 80–89, 2009.

[16] J. Shen, E. Fitzhenry, and T. G. Dietterich. Discovering frequent work procedures from resource connections. In *IUI '09: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 277–286, New York, NY, USA, 2009. ACM.

[17] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th international conference on Software engineering*. ACM, 2008.

[18] E. F. Tjong Kim Sang. Introduction to the conll-2002 shared task: Language-independent named entity recognition. In *Proceedings of CoNLL-2002*, 2002.

[19] K. Toutanova, D. Klein, C. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *North American Chapter of the Association for Computational Linguistics - Human Language Technologies*, 2003.

[20] M. G. Van Kleek, M. Bernstein, K. Panovich, G. G. Vargas, D. R. Karger, and M. Schraefel. Note to self: examining personal information keeping in a lightweight note-taking tool. In *Proceedings of the 27th international conference on Human factors in computing systems*. ACM, 2009.