SmartGroups: Focusing on Task-Relevant Source Artifacts in IDEs

In: ***19th International Conference on Program Comprehension*, 2011**

David Röthlisberger
Software Composition Group
University of Bern, Switzerland

Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland

Stéphane Ducasse
INRIA-Lille Nord Europe
France

Abstract—**Navigating large software systems, even when using a modern IDE, is difficult, since conceptually related software artifacts are distributed in a huge software space. For most software maintenance tasks, only a small fraction of the entire software space is actually relevant. The IDE, however, does not reveal the task relevancy of source artifacts, thus developers cannot easily focus on the artifacts required to accomplish their tasks. *SmartGroups* help developers to perform software maintenance tasks by representing groups of source artifacts that are relevant for the current task. Relevancy is determined by analyzing historical navigation and modification activities, evolutionary information, and runtime information. The prediction quality of *SmartGroups* is validated with a benchmark evaluation using recorded development activities and evolutionary information from versioning systems.**

Keywords: development environments, development activity analysis, task representation, software navigation, software maintenance, dynamic analysis

I. Introduction

To maintain a software system, developers typically use a development environment (IDE) to navigate the system and to locate important artifacts relevant for a particular task, for instance a method which introduced a defect. Navigation is a crucial prerequisite to identify and comprehend the source entities relevant for a software maintenance task [1], [3]. The navigation of a large system in an IDE, however, is a time-consuming activity as there are many source artifacts such as packages, classes, or methods that implement the system [11]. Identifying task-relevant artifacts is further exacerbated by the fact that source artifacts are interconnected with each other at runtime in ways that are hard to foresee while browsing the software space [4], [2], [22], [18]. The set of task-relevant entities, the working set, is usually just a subset of the entire software space. The IDE, however, does not offer a view specifically tailored to the current task to enable developers to just focus on the task-relevant artifacts. Instead, the development environment provides a view on the complete, probably huge software space, which forces developers to navigate forth and back in a complex space.

To determine the severity of these navigational difficulties while performing software maintenance tasks on object-oriented systems in the IDE, we analyzed twenty recorded development sessions of six different software developers performing defect correction and feature implementation tasks in small and medium-sized applications written in Squeak[1] or Pharo[2] Smalltalk. We opted to analyze Smalltalk IDEs instead of wide-spread IDEs for Java such as Eclipse[3] because of the availability of a framework gathering IDE usage data (*e.g.*, SmallBrother[4]) and of developers willing to provide us with such data. As indicators for navigation difficulties we consider the number of *re-visits of source artifacts* purely for reading and understanding (without modification), the *edit/navigation ratio* (ratio of edit actions compared to navigation actions; as navigation actions we consider clicking on any source artifact such as a class, a method, or a package while an edit action is the modification of any such artifact), and *the average extent of navigation between two edits* (how many navigation actions occur between two subsequent modification actions). Table I presents the results of this analysis.

Indicator	Arithmetic mean	Variance
Number of entities revisited	35.10	10.83
Edit / navigation ratio	9.51%	3.21%
Number of navigation actions between two edits	19.31	8.22

Table I
THREE INDICATORS HIGHLIGHTING NAVIGATION ISSUES OCCURRING IN THE SMALLTALK IDE AFTER ANALYZING 20 DEVELOPMENT SESSIONS.

These results corroborate the hypothesis that navigating the source space in an IDE is often difficult. The low edit/navigation ratio (less than ten percent) indicates that locating an artifact to be modified in order to carry out a software maintenance task requires developers to perform many navigation actions. Another indication for ineffective navigation in IDEs is the average number of navigation actions performed between two subsequent modification actions; on average developers perform 19 navigation actions until they again modify an artifact, which we consider to be a large amount of navigation between two consequent modification activities as such extensive navigation is a sign of a missing focus on the relevant artifacts that have to be modified to achieve a task.

We propose in this paper the inclusion of the concept of *working context* in the IDE. A working context is a set

[1]http://www.squeak.org
[2]http://www.pharo-project.org
[3]http://www.eclipse.org
[4]http://www.squeaksource.com/SmallBrother

of artifacts relevant for a particular task. To identify these relevant entities, we define types of tasks, namely defect correction, feature implementation, and general program understanding tasks. Different types of tasks have different relevant artifacts, thus the procedure to identify relevant source elements is dependent on the nature of the task. For defect correction tasks, for example, we also take into account evolutionary information, that is, artifacts that were committed to the source repository in the past to correct a defect. The execution of defective features often gives additional information to find a cure for the problem, thus we also exploit runtime information to identify task-relevant artifacts. For program understanding tasks, mainly navigation activities performed in the IDE are analyzed to suggest relevant entities. Developers manually specify the nature of the task; this information is used to associate development activities with a task type and to recommend based on these past activities the artifacts relevant for future tasks of the same type.

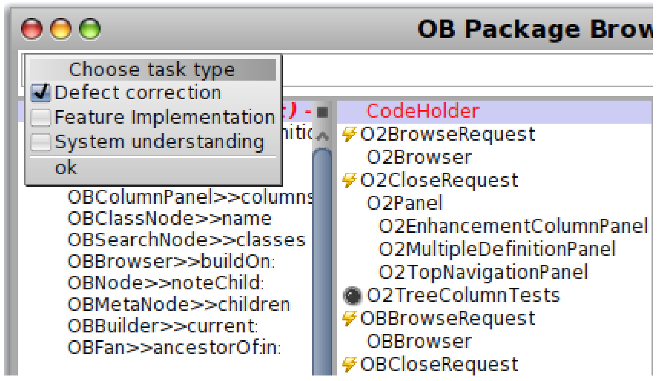
We implemented our proposal as an IDE extension called *SmartGroups* which is available for Squeak and Pharo Smalltalk. This extension represents working contexts by categorizing source entities in groups. These groups are "smart" in the sense that they hold source entities automatically categorized by algorithms tailored to specific task types (defect correction, feature implementation, or system understanding).

The main contributions of this paper are (i) an empirical analysis of the difficulties in navigating the software space in IDEs, (ii) an implementation of *SmartGroups* mitigating these difficulties, and (iii) the validation of how accurately *SmartGroups* identify task-dependent entities.

This paper is structured as follows: Section II introduces *SmartGroups* in a nutshell while Section III reports on related approaches such as Mylyn and NavTracks. In Section IV, we describe *SmartGroups* in detail, that is, the algorithms and their parameters *SmartGroups* use to identify task-relevant source artifacts. Section V evaluates the precision and recall of *SmartGroups* with a benchmark validation based on a recorded set of development activities. Ultimately, Section VI concludes the paper.

II. *SmartGroups* in a Nutshell

When a developer has to correct a defect in a large software system, *SmartGroups* can suggest artifacts that are likely to be relevant for defect correction tasks. For this purpose, the developer specifies in the IDE enhanced with *SmartGroups* the type of task to be accomplished, in this case "defect correction" (cf. Figure 1). Other supported task types are feature implementation and general program understanding tasks. Besides type of the task to be solved, the developer can optionally also limit the search for task-related artifacts to specific packages or can exercise particular features whose execution is analyzed by *SmartGroups* to focus on just those artifacts used in these features.



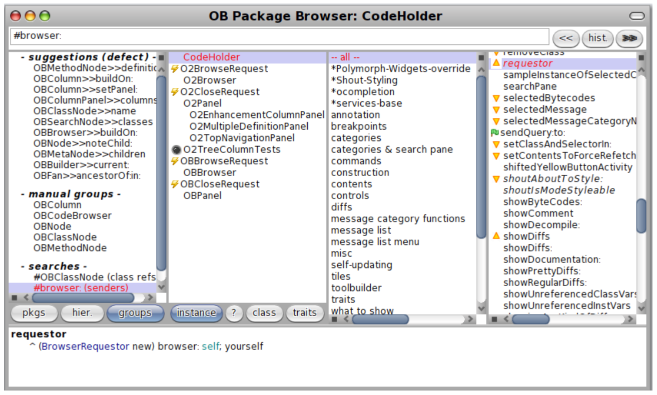
```
OBColumnPanel>>columns
OBClassNode>>name
OBSearchNode>>classes
OBBrowser>>buildOn:
OBNode>>noteChild:
OBMetaNode>>children
OBBuilder>>current:
OBFan>>ancestorOf:in:
```

```
CodeHolder
O2BrowseRequest
  O2Browser
O2CloseRequest
  O2Panel
    O2EnhancementColumnPanel
    O2MultipleDefinitionPanel
    O2TopNavigationPanel
O2TreeColumnTests
OBBrowseRequest
  OBBrowser
OBCloseRequest
```

Figure 1. *SmartGroups* identifies entities related to a task of the selected type.

SmartGroups search for task-related artifacts using the algorithms described in Section IV. The result, that is, the artifacts likely to be relevant for a defect correction task are directly presented in the IDE next to traditional source code views. Figure 2 shows the list of artifacts that are relevant for defect correction tasks and that were identified by analyzing past development activities such as navigation, modification, committing, or the execution of source artifacts.

The *SmartGroups* view is tightly integrated in familiar IDE views. The developer can easily switch between a view focusing on the task-relevant artifacts identified by *SmartGroups* (as depicted in Figure 2) or the traditional view showing the entire software space. This tight integration lowers the burden for the adoption of smart groups presenting task-relevant artifacts. Additionally, source artifacts identified as task-relevant are highlighted in the traditional IDE views to be able to quickly identify them in these views.



```
- suggestions (defect) -
  OBMethodNode>>definitio
  OBColumn>>buildOn:
  OBColumn>>setPanel:
  OBColumnPanel>>columns
  OBClassNode>>name
  OBSearchNode>>classes
  OBBrowser>>buildOn:
  OBNode>>noteChild:
  OBMetaNode>>children
  OBBuilder>>current:
  OBFan>>ancestorOf:in:
- manual groups -
  OBColumn
  OBCodeBrowser
  OBNode
  OBClassNode
  OBMethodNode
- searches -
  #OBClassNode (class refs
  #browser: (senders)
```

```
CodeHolder
O2BrowseRequest
  O2Browser
O2CloseRequest
  O2Panel
    O2EnhancementColumnPanel
    O2MultipleDefinitionPanel
    O2TopNavigationPanel
O2TreeColumnTests
OBBrowseRequest
  OBBrowser
OBCloseRequest
  OBPanel
```

```
-- all --
*Polymorph-Widgets-override
*Shout-Styling
*ocompletion
*services-base
annotation
breakpoints
categories
categories & search pane
commands
construction
contents
controls
diffs
message category functions
message list
message list menu
misc
self-updating
tiles
toolbuilder
traits
what to show
```

```
requestor
sampleInstanceOfSelectedC
searchPane
selectedBytecodes
selectedMessage
selectedMessageCategoryN
sendQuery:to:
setClassAndSelectorIn:
setContentsToForceRefetch
shiftedYellowButtonActivity
shoutAboutToStyle:
shoutIsModeStyleable
showByteCodes:
showComment
showDecompile:
showDiffs
showDiffs:
showDocumentation:
showPrettyDiffs:
showRegularDiffs:
showUnreferencedClassVars
showUnreferencedInstVars
```

```
requestor
    ^ (BrowserRequestor new) browser: self; yourself
```

Figure 2. *SmartGroups* view integrated on the left side of Pharo Smalltalk's system browser, the core of the Smalltalk IDE.

III. Existing Approaches

Several existing proposals also aim at presenting task-relevant entities and at representing a working context in the IDE. However, these related works have several limitations and shortcomings and cannot completely achieve our goal of representing context in the IDE. In the following, we report on these shortcomings of existing work and how we want to overcome them.

FEAT. This approach identifies concerns from recorded program investigation activities performed in the IDE and visualizes these concerns with graphs [15]. However, the quality of the identified concerns is heavily dependent on how organized the analyzed investigation sessions were [14], [16]. Disorganized investigation sessions cannot be used to identify concerns [14], thus FEAT's algorithms are not robust. We tackle this problem in *SmartGroups* by exploiting more than one data source to identify entities belonging to the same context or concern. This renders the *SmartGroups* approach more robust, that is, less dependent on the quality of the analyzed transcript of past investigation activities

NavTracks. This technique recommends source entities related to the currently selected entity by analyzing how developers navigated and modified the system in the past [17]. With *SmartGroups* we take into account more information than just recency of navigation; we also consider evolutionary data (age, versions, or authors of source artifacts) or dynamic data such as number of invocations, memory usage, or execution time. The analysis of this data yields groups of entities that form a particular context, for instance those that are relevant for a specific software feature or that are related to a specific task such as bug correction. These groups are permanently accessible and do not depend on the currently selected artifact, thus they act as a categorization of source entities. The nature of the current programming task is an important factor for the identification of relevant entities. NavTracks recommends related files independently of the task and thus ignores the relation between tasks and importance of entities. This work has been evaluated empirically by observing developers and analyzing their navigation patterns [17].

Mylyn. This proposal exploits programmer activities to build a degree-of-interest model for the program elements in a system and highlights the elements considered interesting for the task-at-hand [8], [9]. *SmartGroups* are related to Mylyn in the sense that they use similar information to automatically build groups of source artifacts, namely recency and frequency of modification and navigation of source entities. However, as mentioned before, *SmartGroups* also exploit dynamic and evolutionary information.

Another difference to Mylyn is that *SmartGroups* adapt to the nature of the development task currently being performed (either defect correction, feature implementation, or system understanding). Depending on the type of task, *SmartGroups* use different algorithms to determine the elements in specific groups. While Mylyn just provides a single and fixed algorithm to identify related entities, *SmartGroups* allow developers to influence how the approach locates relevant artifacts. Developers understand their development task and the system under study usually well enough to support *SmartGroups* in the identification process by, for example, specifying the task type and packages being involved in the task, thus we do not apply such a strict model as Mylyn which computes the degree of interest value for each artifact independently of the nature of the task and the knowledge and experience of the developer. Although developers can alter the elements shown as relevant for the task in Mylyn, they cannot influence how they are initially computed. With a field study the authors have shown that Mylyn can reduce the amount of navigation necessary to conduct software changes [9].

Other researchers also combine different information sources (*e.g.*, information retrieval with execution tracing) to obtain better results to, for instance, locate concerns [5] or bugs [12] in source code.

IV. THE *SmartGroups* APPROACH

To automatically identify source entities relevant for a particular task, *SmartGroups* exploit various kinds of data sources, namely recorded development activities performed in the IDE, evolutionary information extracted from source repositories (versions, authors, etc.), and dynamic information extracted from program execution. All available data sources are combined to reveal task-relevant relations between source artifacts.

By specifying in the interface provided by *SmartGroups* on which type of task the developer starts to work (cf. Figure 1), the developer supports the process of automatically identifying the task-relevant source elements. This task specification is abstract and high-level: the developer can choose between defect correction, feature implementation, and general program comprehension tasks. This task specification can optionally be further refined by enumerating system packages that are relevant for a task or by exercising one or several features with which the task-at-hand is concerned. The developer is encouraged to manually specify when he finished a task to improve the data quality of *SmartGroups*.

As soon as the developer has specified the nature of the current task, *SmartGroups* analyze its various data sources based on the given task specification. Recorded development activities are analyzed with regard to the task type developers performed during the recording. We assume that the same types of tasks involve similar entities; for instance, bug correction is likely to involve certain kind of entities, such as recently added or modified elements [6], elements that contained bugs in the past [20], [7], [10], or that have been frequently changed [6]. Thus, *SmartGroups* specifically take into account recency and frequency of modification to suggest relevant entities for defect correction tasks. Typically, more artifacts are navigated than modified; thus additionally to entities frequently or recently modified we also consider entities that have been frequently navigated but not modified to be task-relevant; the importance of such entities depends on the type of task. To support multiple developers working on the same project, *SmartGroups* can store recorded development sessions in the source versioning system so that all developers can access the same recorded sessions to be used for the suggestion of task-relevant artifacts by *SmartGroups*.

In object-oriented applications developers mostly modify single methods to correct defects or adapt features. For program comprehension, the understanding of methods is also crucial, thus *SmartGroups* mostly suggest methods as task-relevant entities. Classes are also suggested, in particular for program comprehension tasks. As we do not consider the addition of a method to a class as a modification of the class itself (this is not true if attributes are added), classes are rarely modified during maintenance tasks, thus they are usually not directly considered as task-relevant for defect correction or feature implementation tasks.

A. *Task Type Identification*

If recorded development activities do not have a task type associated, *SmartGroups* try to automatically determine this task type by analyzing the recorded activities. A sequence of recorded activities usually contains several distinct development sessions. Start and end of such a development session is either marked manually by the developer, by the termination of the IDE or after a certain period of inactivity (two hours or more). A development session might contain more than one task. For sessions containing commits to a repository, we consider the time of a commit as the end of a defect correction or feature implementation task (see below). Sessions without commits are either considered to be a single program comprehension task, or, if they contain modification activities, as a single defect correction or feature implementation task. Note that the latter two kinds of tasks might also include program comprehension activities; the difference to pure program comprehension tasks is that such kind of tasks also include modification activities. To distinguish between defect correction and feature implementation tasks, we analyze the extent of modification: Sequences of development activities containing just modification actions limited to one or two particular methods or classes are perceived as defect correction tasks; if modification involves several different entities (that is, more than two methods), we assume a feature implementation or adaptation task. This criteria might not always correctly separate defect correction from feature implementation tasks, however, we manually categorized twenty percent of all recorded tasks used in the validation (cf. Section V) by taking into account information given by the developer concerning the task being performed and revealed that this criteria correctly categorized the tasks in all except two cases which corresponds to a precision of nearly 98%, which we consider precise enough for our purposes.

Determining task types from evolutionary data works similar. As this kind of data does not include information about entity navigation, we basically just distinguish between defect correction and feature implementation tasks by considering the extent of modification using the same criteria previously mentioned. If a programmer specifies a type of task in *SmartGroups*, this information is automatically stored in the commit message, thus we can retrieve this information from evolutionary data.

Evolutionary data extends data about recorded development activities (i) by grouping modified entities into a coherent set, that is, the entities being part of the same commit, and (ii) by finalizing a batch of modification actions. From recorded modification actions it is difficult to separate intermittent modifications from those finally solving a particular task. We expect that committed entities contain final changes while recorded modification actions are often just a step towards the final modification of a particular entity in a particular task. We thus take the time of commit as the completion time of a task, at least in cases where developers did not manually specify when a task is finished. Furthermore, commits help to refine information contained in recorded modification activities as they usually just include entities that indeed have to be modified in order to complete a task.

The identification of task-relevant entities based on development activity and evolutionary data works as follows for the different types of tasks:

B. *Defect correction*

First, we map particular commits to recorded development sessions to mark the end of a task. The beginning of a task has been either specified by the developer or has been automatically determined as described above. As soon as the extent of the task in the recorded development activities is determined, we extract all modification actions and the involved artifacts and count how frequently each artifact was modified. The set of modified entities is firstly ordered by frequency and extent of modification and secondly compared to the set of committed entities; modified entities that have not been committed are moved to the end of the ordered list of entities. Additionally, we also incorporate entities that have been frequently navigated but never modified. Such entities are placed at the end of the list, after those not committed. Source elements that have been recently modified or frequently and recently navigated in a development session are considered to be more important and move up in the list.

This procedure is repeated for all defect correction tasks in the recorded set of development activities. The lists of relevant entities from all considered tasks are merged; entities from recent development sessions are prioritized and thus appear higher in the merged list.

As defects often occur in artifacts that have been recently added to the system [6], we increase the priority of artifacts that are young (age is measured in number of commits since an artifact has been initially added to the system). We also rank artifacts higher that have been changed in many

Parameter	**Defect correction tasks**	**Feature implementation tasks**
Initialization	Initially, the list is ordered by extent of modification, that is, number of lines that are added or adapted, and by frequency of modification.	Same as for defect correction
Committed entities	Entities that have been modified but not committed are appended to the end of the list in their initial order.	Same as for defect correction
Frequently navigated but not modified	The 30 most frequently navigated entities are ordered by frequency and appended in this order to the end of the list.	Same as for defect correction, but taking the 20 most frequently navigated entities
Recent navigation	The 100 most recently navigated entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent navigation' list.	Same as for defect correction, but taking the 200 most recently navigated entities
Frequent navigation	The 40 most frequently navigated entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'frequent navigation' list.	Same as for defect correction, but taking the 100 most frequently navigated entities
Recent modification	The 20 most recently modified entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent modification' list.	Same as for defect correction, but taking the 100 most recently modified entities
Recent dev. session	All development sessions are ordered by recency and the weight of all entities in the ranked list is increased by the rank of the development session in which they have been lastly modified.	Same as for defect correction
Age (young entities ranked higher)	The 50 youngest entities are ordered by age (number of commits since creation) in ascending order and for each of these entities appearing in the ranked list we increase its weight by the rank from the 'age' list.	Not used
Number of authors	Each entity is ordered by number of authors in descending order and the weight of each entity in the ranked list is increased by its rank in the 'number of authors' list.	Not used
Same author	Not used	Entities changed by the same author as the current developer are ordered by the recency of the development session in which this author changed the entity. The weight of each entity in the ranked list is increased by its rank in the 'same authors' list.

Table II
THE PARAMETERS USED IN THE ALGORITHM TO IDENTIFY ENTITIES RELEVANT FOR DEFECT CORRECTION AND FEATURE IMPLEMENTATION TASKS AND HOW THEY INFLUENCE THE ORDER OF THE RELEVANT ENTITIES.

commits or that have many different authors, as we expect the likelihood to contain a defect to be higher for artifacts with these characteristics.

The ranked list is shown in the *SmartGroups* view under the label "suggestions" as illustrated in Figure 2. Only the first twenty elements are shown by default. Developers are presented with all elements in the list on demand. We limit the maximum number of entities in the list to 50 elements; elements placed beyond this limit are not presented. Developers can change the position of elements in the list or manually add or remove elements, but the list can never grow beyond 50 elements. This hard limit has been empirically determined by interrogating developers that considered it to be an appropriate compromise between having many suggestions for related artifacts and not overloading the *SmartGroups* view.

The algorithm to rank a specific artifact to determine its position in the list of related artifacts encompasses many parameters such as how frequently navigated entities move up in the list. Each element in the ranked list has an initial weight which equals its position in the list. Each parameter adds weight to some of the entities. We automatically add the maximum weight given by a parameter and increase this weight by one for entities that have not yet received weight for this parameter to move such entities towards the end of the list. Eventually, the list is sorted by the weight of entities in ascending order which leads to the final ranked list.

The parameters in this algorithm are listed and explained in Table II, left column. We empirically determined the optimal value of each parameter by running a benchmark experiment using ten recorded development sessions (the benchmark principle is explained in more detail in Section V). Each session contained several defect correction tasks for which we knew precisely the involved development activities. We used the recorded activities of all but one task to compute the ranked list of relevant entities for the last task in the session. We knew precisely the elements that actually had to be modified to correct the defect of this last task. We then gradually varied in several benchmark runs the parameters in a given range and chose ultimately the parameters from the benchmark run which proposed a list of relevant artifacts best aligned with the set of elements that developers actually had to modify to correct the last defect in each development session. For instance, for the parameter of how many of the most frequently navigated entities should added to the list of relevant entities, we initially started with five entities, increased this value in steps of five, and ultimately revealed that a value of 30 entities yields best results.

C. Feature implementation

The identification of source elements relevant for feature implementation tasks works largely in the same way as described above for defect correction tasks, except that feature implementation tasks extracted from recorded development activity and source code history are analyzed instead of defect correction tasks.

There are some minor differences in the identification algorithms compared to defect correction. For instance, we rank artifacts higher that have been previously modified or navigated by the same author as the current developer, as we consider it to be likely that the same developer will work on similar features throughout the lifetime of a system. Thus, entities this developer changed during previous development sessions are more likely to be relevant for the current task than artifacts this developer has never touched before. We expect this effect to be less pronounced for defect correction tasks as often defects have to be urgently corrected, thus the first available developer may perform the correction and not the one who normally works on the affected feature. Compared to defect correction tasks, we adapted the parameters of the identification algorithm as depicted in Table II, right column.

D. General program comprehension

Identifying source elements relevant for program comprehension tasks differs from the procedure discussed above as this type of task does not encompass any modification, thus we cannot consider evolutionary information or modification activities for the identification process. We only take into account navigation activities for program comprehension tasks. We build the list of related entities in the following way: (i) the initial list is ordered by how often an entity was navigated, (ii) recently navigated entities are ranked higher, (iii) entities which developers selected in the result lists of searches are considered to be more important, (iv) the more time developers spent reading a specific artifact, the more importance we assign to it (an entity's "reading time" is measured in the outlier-adjusted time spent between selecting this entity and selecting the next one), and (v) the longer a view on a particular entity is open, the higher we rank this entity. An entity can still be opened in a view, even though the developer currently looks at another entity in another open window or tab, hence time of visibility is often longer than reading time. Table III depicts the parameters of this algorithm.

For program comprehension, entities added or changed during defect correction and feature implementation could also be highly interesting. The ten top elements appearing in the ranked lists of the two other task types are also taken into account for program comprehension tasks; they either move up in the ranked list of the latter if they have already been identified as relevant for the program comprehension task, or are appended to the end of the list otherwise.

Parameter	Description
Recent navigation	The 100 most recently navigated entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent navigation' list.
Recent dev. session	All development sessions are ordered by recency and the weight of all entities in the ranked list is increased by the rank of the development session in which they have been lastly modified.
Search results	The weight of all entities to which developers have navigated from search results is not increased while the weight of all other entities is increased by ten.
Reading time	All entities are ordered by reading time in descending order and the weight of each entity in the ranked list is increased by its rank in the 'reading time' list.
Time of visibility in a view	All entities are ordered by their visibility time in a view in descending order, and the weight of each entity in the ranked list is increased by its rank in the 'time open in view' list.

Table III
THE PARAMETERS USED IN THE ALGORITHM TO IDENTIFY ENTITIES RELEVANT FOR PROGRAM COMPREHENSION TASKS AND HOW THEY INFLUENCE THE ORDER OF THE RELEVANT ENTITIES.

Parameter	Description
Not used artifacts	All artifacts not used in the recorded execution of a feature are moved to the end of the list. Thus, such entities appear after all used entities in the order they had in the original list.
Frequency	All used entities are ordered by frequency of occurrence in the method call tree and their weight in the ranked list is increased by the rank they have in the 'frequency of occurrence' list.

Table IV
THE PARAMETERS FOR CONSIDERING DYNAMIC INFORMATION TO REFINE THE RANKED LIST OF RELEVANT ENTITIES.

E. Inclusion of dynamic information

Behavioral information is not always available, thus we do not include such information in the basic algorithms identifying task-relevant entities. However, if dynamic information is available it can greatly improve the predictive quality of the algorithms used in *SmartGroups*. To gather dynamic information the developer has to run the software feature(s) to be corrected, adapted or understood. The execution of features is analyzed using partial behavioral reflection [19] which allows us to precisely select which operations of a program should be analyzed. For feature analysis it is enough to only analyze method invocations in application methods and classes, but not in libraries for instance.

The collected dynamic information (basically a tree of method invocations) influences the ranked list of task-relevant artifacts identified based on development activity and source history information in the following ways: (i) the ranking of artifacts not used in the executed feature(s) is decreased and (ii) artifacts appearing several times in the method invocation tree in different branches move up in the list. The parameters used in the algorithm considering dynamic information are depicted and explained in Table IV.

Artifacts appearing in the gathered method invocation tree but not in the ranked list are only appended to the list if it has not yet reached the limit of 50 elements. These dynamically identified entities are added to the list in the order determined by number of occurrences in distinct branches of the call tree. Thus, dynamic information refines the ranked list already identified based on development activity and evolutionary information.

V. Validation

This section validates *SmartGroups* by two means: (i) we evaluate how accurate the suggestions for task-relevant artifacts are and (ii) we report on the practicality of *SmartGroups* by presenting user feedback.

A. *Correctness of* SmartGroups

For the adoption of *SmartGroups* by developers it is crucial that the suggestions for relevant artifacts be accurate, that is, the automatically determined entities supposed to be relevant for the current task should meet the following criteria: (i) the suggested entities should indeed be task-relevant (high precision, few false positives) and (ii) many of the task-relevant entities should be suggested (high recall, few false negatives).

Procedure. To evaluate precision and recall of the suggestions of *SmartGroups* for task-relevant artifacts, we conduct a benchmark validation; benchmark validations have already been used for similar purposes by other researchers such as Robbes *et al.* [13]. We analyze a recorded sequence of development activities (navigation and modification actions performed in the Smalltalk IDE) accompanied with evolutionary information (commits, versions, authors). We automatically identify the task types as described in Section IV because developers did not specify the task types during the development activities we recorded. In an initialization phase, we use the first ten tasks of each type appearing in the sequence of development activities to build the initial lists of recommendations for task-related artifacts. To measure the accuracy of *SmartGroups*, we compare the recommendation list for a particular task type with the set of entities that have actually been relevant for the subsequent task of this type. For example, the ten first defect correction tasks suggest relevant entities for the eleventh defect correction task in the recorded sequence of development activities and the accuracy of the suggestions for the eleventh task is measured. The first eleven tasks are then analyzed to build the lists of relevant entities for the twelfth task, the accuracy of the suggestions for this twelfth task is measured, and so on until the end of the sequence of activities is reached.

Identification of task-relevant entities. For a particular task, we determine the entities that are actually relevant as follows: For defect correction and feature implementation tasks, relevant entities are those that are committed to the source code repository during the execution of a task. For program comprehension tasks that usually do not contain any modifications or commits, we consider all navigated entities to be relevant.

Dataset. The recorded datasets we analyzed in this benchmark stem from five different developers who contributed in total nearly 50’000 navigation and modifications events that were accompanied by 268 commits to a source repository. These developers worked on six different systems of medium size (consisting of between 300 and 1200 classes with an average of approx. 75,000 LOC). All developers are experienced Smalltalk developers with at least four years of programming experience in the Smalltalk environment. During this study these developers were performing their daily programming work in their normal working environment on various kind of systems ranging from developing industrial web-based applications to software analysis environments. These systems were very familiar to the respective developers, in most cases the developers originally developed them. The time span covered in the recorded sets for each system varies from three weeks to five months. For each system, we use the recorded sequences of development activities independently of sequences originating from other systems to evaluate the accuracy. At the end, we average the determined accuracy measured over all available sequences of activities. All datasets have been recorded with the publicly available IDE activity recording framework SmallBrother.

Evaluation. To determine how accurate the identified task-related entities are, we compare the set of entities that have actually been related to the task (determined with recorded development activities and evolutionary information) with the suggestion list of *SmartGroups*. This list is ordered and contains a maximum of 50 elements. None of the recorded defect correction or feature implementation tasks spanned 50 elements (the number of relevant elements varied between one and 37). Actually relevant task entities should be included in the respective suggestion list for each task to achieve a recall of 100%. Some program comprehension tasks exceeded the limit of 50 elements. For these tasks, we temporarily allowed *SmartGroups* to suggest more than 50 entities, namely all elements it could identify as being task-relevant. To measure recall we count the number of task-relevant entities not identified by *SmartGroups* (false negatives). Precision is measured by analyzing how many entities *SmartGroups* suggest that are actually not task-relevant (counting false positives). Precision and recall are computed according to the definitions of Rijsbergen [21].

True positives are the relevant entities *SmartGroups* correctly identified, false positives the entities *SmartGroups* wrongly identified as being relevant, false negatives are the relevant entities *SmartGroups* could not identify. Note that it is not possible to determine the true negatives as we do not know the exact number of source artifacts in the system at any one time during the recorded dataset. Thus, we cannot compute the accuracy of *SmartGroups* defined as the proportion of true results. Precision and recall, however,

Measure	Value
Number of tasks	172
Number of activities	15'364
Number of commits	179
Precision	39.0%
Recall	65.3%

Table V
RESULTS FOR DEFECT CORRECTION TASKS.

Measure	Value
Number of tasks	84
Number of activities	7'982
Number of commits	86
Precision	35.2%
Recall	54.9%

Table VI
RESULTS FOR FEATURE IMPLEMENTATION TASKS.

Measure	Value
Number of tasks	143
Number of dev. activities	21'354
Number of commits	0
Precision	24.9%
Recall	20.7%

Table VII
RESULTS FOR PROGRAM COMPREHENSION TASKS.

give a good impression of *SmartGroups*'s accuracy. High precision and high recall values lead to a high accuracy [23].

These measures are computed for each task individually and are averaged over different tasks by computing the arithmetic mean value.

Results. We show the results of the benchmarks separated by type of tasks. The result tables present precision and recall averaged over all analyzed tasks of a particular type (except the tasks used to initialize the identification procedure). Table V presents the results for defect correction tasks, Table VI for feature implementation and adaptation tasks, and Table VII for program comprehension tasks.

Note that we were not able to use all recorded development activities as some were identified as belonging either to a defect correction or a feature implementation task, but there was no corresponding commit in this time period, hence we could not determine a set of entities actually being relevant for such a task. We skipped such sequences of development activities. For program comprehension tasks it is not necessary to have a corresponding commit. We ignored, however, development sessions that matched the criteria for being concerned with a program comprehension task but which lasted a very short amount of time, that is, a few minutes. In general, we consider the identification of program comprehension tasks as less reliable than for the other two task types.

Result interpretation. The results show that precision and recall for defect correction and feature implementation tasks are fairly high. While a precision below 40% might be considered as low, we need to be aware that a reduction of the number of entities that need to be studied by a developer is already beneficial in itself; even though there are elements in the suggested list that are not accurate (as indicated by the precision of less than 40%), it is much easier for a developer to navigate a selection of entities than the entire system. For that reason we accept some false-positives in the list as long as many of the actually important elements are included (as indicated by a recall value of up to 65%). One reason why recall is not higher is that developers also have to work on bugs or features that are completely unrelated to any tasks that have previously been solved; in such a case *SmartGroups* cannot correctly derive the relevant artifacts from the recorded development history. As closely related proposals such as NavTracks [17] or Mylyn [8] do not indicate their precision and recall, we cannot compare our results. For program comprehension tasks, both precision and recall are rather low. We attribute this to the fact that identifying program comprehension tasks and separating them from other kind of tasks was more difficult than for defect correction and feature implementation tasks. Furthermore, *SmartGroups* have to rely on much less information, basically just historical navigation activities, to determine artifacts related to program comprehension tasks, while for the other types of tasks, modification activities and evolutionary information can considerably improve the prediction quality of *SmartGroups*. We expect that *SmartGroups* yield similar results for program comprehension tasks as Mylyn, NavTracks or FEAT.

Threats to validity. There are several threats to validity in our experiment:

Task type identification. As mentioned above, automatically deferring the type of task from a sequence of recorded development activities is error-prone. We might have mistaken feature implementation tasks for defect correction tasks, and vice-versa. Furthermore, since separating a development session from another one is either based on a large amount of time elapsed between two activities or by terminating the IDE, the same task might actually span more than one development session. However, for program comprehension tasks we assume that they are completed at the end of a development session while the developer actually might have continued with this task in the next session. Similarly, it could be that at the beginning of a session, the developer worked on a program comprehension task unrelated to the defect correction task following afterwards. Yet still the entire session, at least until the first commit ending the defect correction task, is considered to be a defect correction task.

Granularity of tasks. The three task types we propose are very high level. There are several kinds of tasks such as performance optimization or refactoring that do not match any of the three task types. In our experiment, however, such tasks would be considered to be either defect correction or feature implementation tasks. A more fine grained categorization of tasks is more realistic and is likely to also improve the accuracy of the suggestions determined by *SmartGroups*.

Parameter determination. We determined the different parameters and their values (cf. Table II, Table III, and Table IV) used in the algorithms to identify relevant entities for specific types of tasks by running a benchmark validation using ten recorded datasets. These datasets were different than those used in this validation, but partially stem from the same developers working with the same systems as we considered in the validation. The ten datasets stem from three different developers working on four different systems. Two of these three developers also contributed datasets to this validation, and two of the four systems were also covered in the validation. Thus, the determination of the parameters is based on similar development sessions as those we used to validate *SmartGroups*. Nonetheless, we do not expect that this fact imposes a considerable threat to validity as the different development activities and tasks are fairly typical for software maintenance because all of them were concerned with software systems representative in size and complexity for many industrial applications. Thus, we expect similar validation results even if the parameters had been gauged using other sequences of development activities.

Assumption of optimal navigation. For program comprehension tasks, we specify that all entities that have been navigated in the recorded dataset are task-relevant. It is likely, however, that developers did not optimally navigate the system to answer the task-relevant questions as they did not have a perfect knowledge about the system. As the developers whose activities we recorded were very familiar with the respective systems they were working on, we expect their navigation to be effective and close to optimal. An indicator for navigation problems mentioned in the introduction, namely number of entities revisited, was with 16.79 on average lower than in the datasets of developers navigating unfamiliar systems, which makes us confident that the recorded navigation was focused. As no modification occurs in program comprehension tasks, we could not measure indicators like edit/navigation ratio.

Generalization. It is unclear how well the recorded dataset of development activities and tasks are typical and representative for software maintenance. There are several variables that might impose a threat to the generalization of the experimental results, such as the extent or severity of the defects corrected during the recorded tasks, the extent of the implemented or adapted features, the software systems being worked on, the length of the development sessions or tasks, and the developers themselves. Most developers that provided us with recorded datasets are researchers from academia working on research tools. It is impossible to say whether systems and developers from industry would lead to other results when assessing the prediction quality of *SmartGroups* for entities relevant for tasks concerned with industrial software systems, even though the considered systems are fairly representative in terms of size and complexity. Further experiments need to clarify this point. We do not expect the performance of *SmartGroups* to depend heavily on the nature of the system or on the developers maintaining this system. The quality of the recorded datasets on which *SmartGroups* base the prediction of task-relevant entities, particularly for program comprehension tasks, is crucial though. For this reason, recorded navigation of novice developers unfamiliar with a system should not, for instance, be used to predict relevant artifacts.

Conclusions. This benchmark validation showed that the algorithms proposed by *SmartGroups* are able to identify task-relevant entities with a precision ranging from 24.9% to 39%, and a recall ranging from 20.7% to 65.3%, at least for defect correction and feature implementation tasks. The predictive quality for relevant entities, however, drops for program comprehension tasks, which we attribute to the lack of substantial and reliable information to suggest related entities for this type of task.

B. *User Feedback*

From face to face discussions with five developers using *SmartGroups*, we got the following feedback:

Importance of Context. All developers stressed how important a context representation in the IDE is when we showed *SmartGroups* to them. In their daily work, they are overwhelmed with information, particularly with views containing too many static source artifacts. Developers want to be able to focus on artifacts relevant for their current task. For this reason, they considered the various smart groups as very useful. They also appreciated the categorization of search results, but asked for an automatic mechanism to remove old queries from this group as old search results are unlikely to be useful anymore after a while. The developers we discussed with were not very excited about the manual smart groups. They might use them occasionally, but it is usually too much of a burden for them to manually add entities to a smart group and to maintain these groups on a regular basis. They appreciate, however, the fact that such manual groups can be used to communicate important aspects of a system by distributing smart groups containing for instance, important artifacts of a system crucial for its understanding. In general, the ability to distribute smart groups between developers was highly appreciated.

Limited number of presented entities. All developers were glad to be able to focus on a limited number of entities (not more than 50) considered to be task-relevant. They agreed with the principle of ranking the entities by assumed relevance and to show low-ranked entities less prominently, that is, in an extended list, while only the first 20 elements are shown by default. As developers experienced that sometimes the suggested elements did not include those they actually had to modify, for instance, to correct a defect, they expressed the wish to be able to access the complete list of entities considered to be relevant by *SmartGroups*, even the elements ranked after the first 50 elements. In general, developers considered the ranking mechanism as intransparent and thus wanted to see all entities identified as possibly relevant, since the automatic ranking might have wrongly put a related entity after the first 50 elements causing it to be stripped away.

VI. Conclusions

SmartGroups mitigate the problem of being overloaded with information in IDEs by explicitly representing context by means of working sets consisting of a small portion of all source artifacts of a particular system. In particular the automatic identification of task-relevant artifacts supports developers to quickly locate artifacts of importance for a particular defect correction or feature implementation task. Developers have to spend less time navigating the software space as *SmartGroups* provide them with a suggestion list of relevant artifacts on which they can focus. As revealed by empirically validating *SmartGroups* by means of a benchmark validation, the automatic determination of task-relevant entities provides a precision of 39% and a recall of 65.3% for tasks encompassing modification activities and commits, but it is more error-prone for pure navigation tasks performed to gain an initial understanding of an unfamiliar system.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010)

References

[1] V. Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1):3–12, 1997.

[2] M. C. Chu-Carroll, J. Wright, and A. T. T. Ying. Visual separation of concerns through multidimensional program storage. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 188–197, New York, NY, USA, 2003. ACM Press.

[3] T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[4] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[5] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Gueheneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proceedings of ICPC 2008*, pages 53–62. IEEE Computer Society, 2008.

[6] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

[7] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of ICSM 2005*, pages 263–272, 2005. IEEE Computer Society.

[8] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[9] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM Press.

[10] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[11] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of ICSE 2005*, pages 125–135, 2005.

[12] D. Poshyvanyk, A. Marcus, G. Antoniol, and V. Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of ICPC 2006)*. IEEE Computer Society, 2006.

[13] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE 2008*, pages 317–326, 2008.

[14] M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–234, Oct. 2003.

[15] M. P. Robillard and G. C. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of 25th International Conference on Software Engineering*, pages 822–823, May 2003.

[16] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1):3, 2007.

[17] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *Proceedings of ICSM 2005*, pages 325–335, 2005. IEEE Computer Society.

[18] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *Readings in artificial intelligence and software engineering*, pages 507–521, 1986.

[19] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

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

[21] C. v. Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[22] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

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