The SEXTANT Software Exploration Tool

Thorsten Schäfer, Student Member, IEEE, Michael Eichberg, Michael Haupt, and Mira Mezini

Abstract—In this paper, we discuss a set of functional requirements for software exploration tools and provide initial evidence that various combinations of these features are needed to effectively assist developers in understanding software. We observe that current tools for software exploration only partly support these features. This has motivated the development of SEXTANT, a software exploration tool tightly integrated into the Eclipse IDE that has been developed to fill this gap. By means of case studies, we demonstrate how the requirements fulfilled by SEXTANT are conducive to an understanding needed to perform a maintenance task.

Index Terms—Software exploration, program comprehension, reverse engineering, software maintenance, software visualization.

1 INTRODUCTION

SOFTWARE maintenance is the most complex and costly phase in the development life cycle [1]. In this process, an understanding of the code to be changed is crucial [2], e.g., to ensure that the intended behavior of the system is not broken [3]. This explains the importance of tool support for software comprehension [2], of which two categories can be distinguished:

- Software visualization tools make use of sophisticated visualization techniques to amplify cognition [4], [5], [6] (e.g., [7], [8], [9], [10], [11]).
- Software exploration tools provide basic navigation facilities, i.e., searching and browsing (e.g., [12], [13], [14], [15], [16], [17]).

The boundary between these two categories is fuzzy. Visualization tools often have limited navigation means, and exploration tools need to visualize the exploration results.

The study presented in [18] emphasizes the key role of navigation for an effective comprehension process. Developers “work through source code” by searching for elements and by browsing along the elements’ relations [19]. During this process, they construct a mental map of the explored information [20]; an appropriate visual representation of this map is critical to prevent developers from getting lost [14]. The category of tools with support for navigation and for explicit representation that eases the construction of a mental map, which we call visual exploration tools, is the focus of this paper:

1. We discuss and justify five functional requirements on visual exploration tools and observe that existing tools only support a subset of them.
2. We present SEXTANT [21], a software exploration tool with support for all five requirements and discuss how different building blocks of its architecture contribute to the fulfillment of individual requirements. This discussion provides useful insights for the design of visual exploration tools.
3. We present four case studies involving tasks typically occurring during software maintenance, in the context of which we compare the comprehension process using SEXTANT with those using two other comprehension tools. The experiments show that the proposed features are crucial for an effective software comprehension.
4. We discuss SEXTANT with respect to three lists of requirements from papers discussing comprehension support with respect to cognition [17], [25] and functional/nonfunctional requirements [23], respectively.

The remainder of this paper is structured as follows: Section 2 elaborates on the functional requirements we impose on software exploration tools. Section 3 presents SEXTANT from a user’s point of view. Section 4 presents the architecture of SEXTANT. Case studies are described in Section 5, while the discussion of SEXTANT with regard to other requirements is presented in Section 6. Section 7 lists open issues for future research. Related work is discussed in Section 8. Section 9 summarizes the paper.

2 FUNCTIONAL REQUIREMENTS

We analyzed research work on comprehension tools as well as user studies and extracted five functional requirements that are of utmost importance for the comprehension process during software exploration. In the following, we justify each requirement by reporting on evidence for them in other work. We also sketch typical scenarios where the required features are needed.

2.1 R1: Integrated Comprehension

Three basic software comprehension strategies have been described [24]: bottom-up, top-down, and mixed strategies. Developers using a bottom-up approach [25] start with the low-level source code and abstract stepwise from it. Developers following the top-down approach [26] use domain knowledge to formulate an initial hypothesis about the software, which is then “verified” by searching for corresponding structures in the code. Mixed strategies imply that both aforementioned strategies are available [27], [28]. Von Mayrhauser and Vans [2] present the integrated model of software comprehension, a refined mixed strategy, in which developers use bottom-up and
top-down approaches at different abstraction levels, frequently switching between them.

The need for integrated comprehension has also been identified in user studies. Storey et al. [24] compare three tools for software exploration and observe that the comprehension strategy is chosen based on various factors such as the program to be understood, the characteristics of the task to perform, and the programmer’s experience. Another exploratory study of expert programmers [29] shows that developers often start with a top-down strategy, but finish with a bottom-up strategy.

Based on these findings, we require that visual software comprehension tools should support an integrated comprehension subsuming both the bottom-up and the top-down approach. Yet, most existing tools impose a fixed strategy.

### 2.2 R2: Cross-Artifact Support

Most exploration tools support only a single kind of artifact, namely, source code. However, modern software projects, e.g., those based on persistence frameworks [30], or component technologies [31], involve a variety of different types of documents, including source and binary code, XML deployment descriptors, scripting, and configuration files. Moreover, the information stored in different documents is tightly interrelated [32]. Hence, it is no longer possible to analyze and comprehend a software system without considering and aggregating information contained in various kinds of artifacts. This also holds for systems that use different programming languages, where the code in each of the languages represents a particular kind of artifact.

The need for cross-artifact navigation has been identified in the context of a field study during the corrective maintenance of a large-scale software system [33]. In this field study, requirements on tool capabilities were derived based on developers’ information needs; the most important ones concern navigation over arbitrary software artifacts.

We conclude that exploration tools enabling software comprehension across artifact borders are required. Indeed, recent tools have adopted support for several kinds of artifacts. For instance, GSEE [13] supports “multisource exploration.” Harrison et al. [34] also recognize the need for representing various types of artifacts.

### 2.3 R3: Explicit Representation

Many visualization tools, e.g., Rigi [9], or SHriMP [10], communicate a mental map of the overall program structure [24]. Software exploration tools must also provide an explicit map of the exploration path. Developers explore software to understand which elements are relevant for a given task and how those elements are related [35]. Each of those navigation steps needs to be represented explicitly; otherwise, it is hard to build a mental map, which often results in getting lost during the exploration process [14].

In Eclipse [36], for instance, it is possible to use hyperlinks to navigate from one software element to another. One can navigate from a class to its superclass and from a method to its callees, and so on. However, the path followed during such an exploration is invisible. By using specialized views like the call hierarchy view, it is possible to see the path for a single kind of relationship. However, switching to a different kind of relationship requires switching the view, potentially causing disorientation [22].

In a user study, Storey et al. [24] observed that representations of previously visited elements acted as important navigational cues. Singer et al. [17] observed that developers often forget details about a particular area of a system when they move to other parts. To recall this knowledge, they have to reexplore previously visited elements. In the study by Sillito et al. [37], Java programmers were observed during change tasks using the Eclipse IDE. The study shows that entities and relationships are often revisited; however, reexploration was not always straightforward, due to lack of an explicit representation of already visited elements. The authors suggest that tools should provide a notion of visited entities and relationships.

The visual representation of the navigation path can also be used for reasoning [38]. However, many tools make reasoning difficult due to lack of referential integrity: The same element may appear several times, potentially in different views of the tool. Let us assume, e.g., that a method $m_1$ is called by methods $m_2$ and $m_3$. In Eclipse, one can navigate from $m_2$ and $m_3$ to $m_1$. However, in this process, $m_1$ will appear twice in two apparently unrelated views, i.e., the referential integrity of $m_1$ is not maintained. This makes it hard to discover that $m_2$ and $m_3$ are related by the common callee $m_1$. Such “implicit relationships” are difficult to integrate into the constructed mental map [24].

A representation that makes explicit all relevant information, including various kinds of relationships, serves as externalized information and relieves developers from memorizing all details [37]. This enables them to build a more accurate and comprehensive mental model. Hence, the authors of [37] suggest views that show more information simultaneously including multiple relationship types.

In summary, to support the creation of a mental map of the software, all of its elements and the relations between them should be visualized explicitly in a single integrated view. Further, the referential integrity among them should be maintained as the navigation process unfolds.

### 2.4 R4: Extensibility

Developers often navigate to common software elements and browse along common relations. For instance, during the exploration of a Java application, developers often navigate to classes, methods, and fields using common relations such as inheritance and method calls. However, specific application domains and/or libraries may necessitate navigation to new kinds of elements and along new kinds of relationships. When exploring an EJB project, e.g., technology-specific relations between a class, its corresponding public interfaces, and related elements in the deployment descriptors become relevant relations to navigate along.

However, tool developers cannot foresee all contexts in which an exploration tool will be used and current tools often lack support for extensibility. Sillito et al. [37] observed that several questions posed by developers were not easy to answer due to lack of proper support by the respective tools in use. When using Eclipse for example, a developer could not navigate along relations specific to a particular model-view-controller framework. Furthermore, Ko et al. [35] report that navigation is often needed along “indirect” dependencies such as the dependency of a variable’s use on the method that computed its most recent value. Current IDEs, however, lack means of indirect navigation.

Several researchers have recognized the limitations of tools with a fixed set of queries. Robitaille et al. [15]
identified the need for design-level queries and Favre [13] claimed that tools should allow experienced users to customize them. In the same vein, we require that software exploration tools should be extensible by domain-specific navigation elements and relationships as well as by means for navigating along indirect relations.

## 2.5 R5: Traceability

Mostly, comprehension is not an isolated development task. Rather, it often serves a follow-up source code modification [39]. Further, the exploration process might only give us hints; the actual understanding or the verification of a hypothesis may require reading the source code [11]. Thus, traceability, i.e., the ability to switch seamlessly between the graphical notation and the corresponding source code, is essential for practical use [18].

### 3 Sextant in Action

In this section, we go through an example scenario to present SEXTANT from a user’s point of view. All features described and highlighted in Fig. 1 are contributions of SEXTANT, except for Eclipse’s Java editor (index 9 in Fig. 1). In the scenario, a developer who is interested in all elements related to the timing concern in a telecommunication application starts using SEXTANT by selecting the project to explore in the combo box (index 1).

First, based on domain knowledge she searches for all types that have the term time in their names: The search query (“Find type using regular expression”; index 2) is selected with a corresponding name pattern as its input. A single node representing the class Timer appears (index 3).

Next, the developer uses this element as a starting point to explore further elements of the timing concern. A right click on the Timer node shows a context menu with all queries applicable to Java classes (the context menu for methods can be seen at index 4). She chooses the query “declared methods” which yields three new nodes (indexes 5a to 5c) representing the methods declared by Timer. For simplicity, the captions for relationships were left out; SEXTANT supports different representations for different kinds of relationships.

The exploration process continues with the start method, following the call hierarchy of this program element. The developer recursively uses the called by relation to investigate the use of Timers. Three new method nodes are explored, namely, complete (index 6), pickup (index 7), and a method also named pickup that takes a parameter.

The developer activates the synchronization feature (index 8) to navigate from the graphical representation to the source code. After clicking on the pickup node (index 7), the editor shows the corresponding source code fragment (index 9), an investigation of which reveals that this method is not part of the timing concern; thus, it is hidden from the exploration graph (index 10).

The exploration continues with the remaining two methods declared by Timer, namely, getTime and stop (index 5c and 5b, respectively). The resulting graph directly shows that both methods are called by the method drop (index 11).

The example scenario illustrates the following features of SEXTANT:

- Bottom-up and top-down comprehension are supported (R1) by two navigational styles: searching and browsing. As many other comprehension tools [40], SEXTANT follows a
query-based approach both to search for program elements and to browse along different kinds of relations.

The results of each search or browsing step are shown in a single graph-based view (R3). This enables switching between searching and browsing without losing the context. Further, it is possible to browse along different kinds of relations without switching views. The graph-based view explicitly shows all explored elements and relations while maintaining their referential integrity.

The synchronization feature enables switching from a visual element to its corresponding source code fragment (R5). This facilitates detailed reasoning about the program: The source code representation is an alternative view providing low-level information. Furthermore, the developer can instantly change the code without having to navigate to the corresponding locations manually. Currently, the visualization is not updated automatically when the source code changes; developers can request an update manually.

Finally, one can hide irrelevant elements and relations from the exploration graph. If a developer uses this feature, the final exploration graph represents a working set with all relevant elements and relations.

Another important feature, not illustrated by the example scenario, is that developers can add new queries to extend the built-in navigation means (R4). It is even possible to specify queries that span across different kinds of artifacts (R2).

4 Architecture

SEXTANT’s architecture consists of four building blocks (see Fig. 2). Important decisions on their design follow from the functional requirements outlined in Section 2; Table 1 shows how each building block relates to functional requirements R1 through R5.

In the following, we briefly elaborate on these relationships before presenting each building block in more technical detail, the rationale being that implications of the requirements R1 to R5 on the architecture apply to any software exploration tool that supports them.

Requirements R1, R2, and R4 influence the design of the data access and navigation facilities. Any exploration tool enables developers to navigate through the software space. As already argued, software is often comprised of various kinds of artifacts (R2) and tool developers cannot foresee all relevant kinds (R4). To satisfy these two requirements, a generic store for software data, capable of storing information about an extensible set of heterogeneous artifacts, is needed. Furthermore, for an integrated comprehension strategy (R1), the data access and navigation facilities must support both searching and browsing over a data set [16]. These two navigational styles can be based on querying [40]. Hence, the data access and navigation facilities should also provide means to access artifact-specific information from the software store.

During exploration, typically a small subset of program elements and relations is revealed. Storing and managing this subset in a way that abstracts from the underlying data is the responsibility of the internal software model. The design of this building block is influenced by R3 and R4. Because of R3 (referential integrity), each explored element and relation may appear only once in the internal model. To support R4, the internal model must be easily extensible with new kinds of elements and relations.

The explored elements and relations are visualized in the graph-based integrated view. For the sake of integrated comprehension (R1), a single view should show all information revealed in a navigation step. Further, with a graph-based view, the structure of the visualization matches the structure of the internal software model and referential integrity is maintained in a straightforward way (R3).

Finally, requirement R5 necessitates that exploration tools are integrated into an IDE to enable developers to use an environment they are familiar with and to reuse existing tools such as code editors.

In SEXTANT, the first three building blocks are implemented as a set of plug-ins integrated into the Eclipse IDE. In the following, we elaborate on the implementation of each of them.

4.1 Data Access and Navigation

The responsibilities of the first building block are 1) to provide access to the raw data representing the software to explore and 2) to provide querying facilities. SEXTANT builds upon the cross-artifact information engineering platform Magellan [32]. (See the bottommost block in Fig. 2. In [32], the tool was called XIRC.)

Data storage and basic access: In brief, Magellan transforms all sources of a project, e.g., Java code, configuration files, or XML descriptors, to a single common format, namely, XML, and stores them in a database. For each kind of artifact, a so-called input processor is responsible for the transformation.

<table>
<thead>
<tr>
<th>Building block / Requirement</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data access and navigation</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internal software model</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph-based, integrated view</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDE integration</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>
Magellan provides an extension point to plug in new input processors for various kinds of artifacts (R4), including programming languages.

The unification of all artifacts into XML enables the use of a single query language to relate information that is spread over different types of artifacts (R2). We use XQuery \cite{41}, a declarative, functional query language especially well-suited for XML data sources. Details on the representation of Java code in Magellan as well as on the specification of queries can be found in \cite{32}. For this work, we extended Magellan with three types of queries.

Search queries enable searching the entire database for program elements. They return references to the nodes of the XML database that match the selection criteria. A search query could, e.g., return all fields with a user-specified name.

Context-dependent queries are defined relative to a previously selected node. By enabling navigation relative to a context, such queries are the key to support the browsing navigation style. For example, given a search query which returns a class node, we can set this node as the context for a query to get all supertypes.

Queries for derived information return information that is not directly stored in the database for a given context. For example, a query that calculates the depth-of-inheritance metric returns derived information.

Each query is associated with a descriptor that contains meta-information, including the name of the query in the context menu of a node, the type of the context nodes for which the query is defined, and the path where the query is stored. The descriptor enables the integration of user-defined queries.

SEXTANT provides more than 40 built-in queries for navigating through Java software, including EJB applications. Developers can add new search queries, new kinds of relations to browse along, and provide new queries for derived information. Immediately after defining and storing a query and its descriptor in a specific folder, the new query is detected and made available.

\subsection{Internal Software Model}
For the internal representation, a generic graph-based model that abstracts from the raw data is used (the middle block in Fig. 2). For instance, in our approach, Java code is represented in XML and, thus, has a tree-based structure. SEXTANT automatically transforms this tree representation into the internal graph-based model.

A node of the graph represents either a software element in the data store or derived information returned from query executions. A node of the first kind has a reference to the corresponding element in the data store; this ensures that each element of the database is represented by at most one node in the graph, even if selected by multiple queries (R3).

The use of generic graph nodes that are typed by referencing their underlying data also facilitates the introduction of new element types (R4). The developer only needs to make a new kind of element available by means of a declarative query; optionally, meta-information can be associated to provide a meaningful description and visual representation, but there is no need to adapt the internal software model in any way.

Relations between two nodes exposed by executing a query are represented by edges in the graph. For each edge, the source is the node that was set as the context for the query evaluation; the target nodes are those returned by the query. Each edge also references the executed query which represents its type. Hence, the semantics of an edge is solely determined by the query; developers can add new queries (together with their meta-information) declaratively to make domain-specific relations accessible in SEXTANT (R4).

\subsection{Graph-Based Integrated View}
Integrated comprehension (R1) is enabled by an integrated view (the topmost block in Fig. 2): Elements and relations discovered in any search or browse activity are visualized by a single graph as depicted in Fig. 1. Elements are represented as nodes, and relations as directed edges. This representation enables switching between searching and browsing. Further, different kinds of relations that represent different views of the software are fused into a single view.

The representation is also crucial for the explicit representation for two reasons (R3). First, if developers browse through a software and discover new program elements, the exploration graph refines the kinds of relations that revealed them. Second, the graph-based nature also ensures referential integrity. Thus, for a given node, a developer can directly see which of the explored relations affect the corresponding program element.

\subsection{IDE Integration}
SEXTANT is tightly integrated into the Eclipse IDE. This is important for the synchronization of the sources of a software project. Whenever Eclipse emits a resource change event as an artifact is added, removed, or changed, the Magellan database is updated accordingly. Further, the integration also affects the visualization layer insofar as we provide means to switch seamlessly between the graphical notation and the source code representation (R5). This enables developers to navigate from the graphical representation of a program element to its corresponding location in source code, which is shown in the Eclipse editor.

In general, the tight integration of tools into an IDE has two further benefits, which carry over to SEXTANT. First, the use of a similar look and feel, e.g., similar context menus for a given program element or the use of similar icons, reduces the learning effort. Second, switching between different tools inside the IDE is more seamless than switching between an IDE and external tools.

\section{Case Studies}
To analyze the effect of the features required in R1 to R5 on the comprehension process, we undertook four exploratory case studies involving five participants. Each study focuses on a typical comprehension task during corrective or perfective maintenance. We favored multiple studies over a single holistic study in order to alleviate the effect of the chosen tasks and subject systems on the comprehension process and to determine whether the data is corroborated between the studies.

Table 2 shows which requirements are discussed in each study. Taken together, the studies illustrate the need for all requirements, R1 to R5.

In the following, we describe the setup of the case studies and shortly describe the other comprehension tools used. Next, we elaborate on the task of each case study and report on the insights gained. Finally, we synthesize the findings and describe how we limited threats to validity.
5.1 Setup

For the experiments, three software comprehension tools with different features were used. All participants performed all four tasks. Tools were assigned to participants so that every participant used each tool at least for one task, as shown in Table 3.

We monitored how the participants built their mental maps by navigating through the subject systems. Further, we asked the participants to think aloud. Since thinking aloud is likely to affect user performance, we did not take time measurements but evaluated the comprehension process qualitatively. We asked the participants to use the features of the comprehension tools during exploration whenever possible to minimize the effect of experience with the navigation features of the IDE on the results of the experiment.

Tasks and subject systems. Four comprehension tasks derived from real maintenance tasks were performed on three different subject systems. All tasks involved only a subset of all software elements and relations, mostly scattered over several modules. The subject systems covered different domains and sizes. Domains range from an EJB application to a virtual machine. The smallest application was approximately 1,000 lines of code; the largest was roughly 200 times as large.

Tools. We compared SEXTANT with two other software comprehension tools, namely, JQuery (version 3.1.1) [14] and Creole (version 1.3.0) [42].

Creole [42] is a plug-in that integrates the SHriMP tool [10] into the Eclipse IDE. It uses advanced visualization to explore Java software, which is represented in a single view as a nested graph. Packages, classes, or methods are visualized as nodes, while relationships between them are shown as edges. Nested elements are shown as nested nodes (e.g., a package which contains five classes is shown in Fig. 3, index 1).

One can collapse/expand parent nodes, e.g., to show the methods declared by a class (index 2). This enables both visualizing software at a high-level of abstraction and revealing low-level details of interest. Elements and relationships can be filtered to reduce complexity. Further, visualization techniques such as fish-eye view or pan and zoom are available.

JQuery [14] is an Eclipse plug-in that combines the advantages of query-based and hierarchical browser tools. Queries are used to search for elements in the system, to create predefined views, or to explore code in terms of various kinds of relationships. The results are visualized in a tree-based, hierarchical representation (index 1 in Fig. 4). Each resulting element can be used as the source for a new query, the results of which build a subtree of the source element (index 2).

The rationale for our choice of tools is as follows: To assess the need for each requirement, we need at least one tool that supports the corresponding feature and another that does not. Both tools support software exploration, but with a different focus: Creole focuses on representation, while JQuery focuses on querying. Table 4 shows that the
tools provide distinct features. Further, Creole and JQuery provide few but essential features. This avoids the potential of disorientation due to “feature overload.”

Finally, for an objective comparison, the same tasks and subject systems must be used. Because SEXTANT currently only supports Java, we needed tools capable of exploring Java software. Many other exploration tools either do not fulfill this requirement or do not provide features distinctive from Creole and JQuery. We tried to include GSEE [13] in the evaluation but, according to the tool’s author, a running version of this tool is no longer available.

**Participants.** Five developers volunteered to participate in the case studies: three PhD students and two master students in computer science. All had reasonable programming skills (≥ 5 years) and good knowledge of Java (≥ 3 years) and Eclipse (≥ 1 year). None of the participants had used the exploration tools before.

**Procedure.** The studies were performed in the lab, in five sessions of 120 minutes, one per participant.

In a first phase of 5 minutes, we asked the participants to fill out a questionnaire covering their background. We briefed them about the purpose of the case study and asked them to perform as well as possible. Next, a training phase of 45 minutes introduced the participants to the comprehension tools. The training consisted of a short demonstration followed by a tutorial, which presented each tool’s features. The tutorial covered a small example application and a simple task that was distinctly different from the tasks in Table 2.

We restricted the training to features necessary for the comprehension tasks to avoid overloading the participants. For instance, we did not show how to open predefined views or how to customize the data extraction. For JQuery and SEXTANT, a short introduction to the query language was given and a set of example queries was provided.

The actual comprehension tasks were performed in 60 minutes using PCs with 20-inch LCD displays. Participants were provided with an Eclipse IDE (version 3.0) pre-configured with the tools and one project for each task. During the tasks, we recorded screen capture videos. Further, the experimenter recorded observations and comments of the participants during the study. Subsequently, about 10 minutes were spent to informally interview the participants about, among other things, the confidence in their understanding, the features they found important, and the features they missed.

5.2 Analyzing a Bug in an EJB Project

**Task.** This study investigated how developers explore software during a corrective maintenance task. The subject was a small EJB application with a bug in transaction management. The bug occurred because a bean’s method which should always run in a transaction context called a method which did not support transactions. The participants received a bug description, a stack trace, and the error message of the application server, exhibiting that a certain component had problems with transaction handling. To find the bug, they had to understand 1) the control flow of the component’s implementation and 2) the transaction attributes of the called methods.

**Observations.** All but participant P5 (using JQuery) found the bug. All participants first searched for the component’s implementation. However, the process of participants P1 and P2 using SEXTANT differed from the others. P1 and P2 applied a domain-specific search query to find the bean implementation based on its EJB name. Next, they used queries to understand the bean’s transaction settings and the control flow of its methods.

P1 and P2 only used SEXTANT; they used EJB-specific queries and did not read the deployment descriptor manually. They produced the exploration graph shown in Fig. 5, which reveals that the method `getValue`, which does not support transactions (edge `TA-NotSupported`) is called in the control flow of the method `getText`, which always runs in a transaction context (edge `TA-Required`).

Since Creole and JQuery lack support for EJB-specific queries, P3, P4, and P5 looked up the class name of the bean in its deployment descriptor, which they then used to search for the corresponding implementation. Next, they tried to understand the control flow of `getText`, which was part of the provided stack trace. During this exploration, they often switched between the tool and the deployment descriptor to look up transaction attributes. In each such navigation step, they tried to integrate the gained knowledge into their mental map. Due to the lack of explicit relations between elements of the deployment descriptor and elements of the Java code, multiple lookups of the same information were observed.

**Results.** We conclude that R2 and R4 are very useful for understanding software involving heterogeneous artifacts. To navigate EJB-specific software structures, Java code and EJB deployment descriptors were needed (R2). Extending the navigation facilities by EJB-specific queries in SEXTANT was very useful to explore EJB-specific elements and relations (R4). Even though JQuery is extensible, the lack of support for R2 makes it impossible to write corresponding queries. This reveals a correlation between R2 and R4: The inability to retrieve information across artifacts makes it impossible to define domain-specific queries that rely on such cross-artifact information.

In addition, the case study demonstrates the need for searching and browsing across different levels of abstraction (R1). Tools lacking these features (Creole and JQuery were of limited usefulness. There was no support for switching comprehension strategies, e.g., searching for the transaction setting of a method during the exploration of the call graph. The users of such tools were forced to lookup and integrate this information manually. In SEXTANT, on the other hand, it was possible to integrate high-level information, such as all methods with specific transaction settings, with low-level information, like the call graph.

5.3 Finding Refactoring Opportunities

**Task.** The subject of this study was Steamloom [43], an extension of the Jikes Virtual Machine [44] with aspect capabilities. Steamloom is a large system of 200,000 lines of code. The task was to analyzie the internal structure of the dynamic weaving module, one of the most complex in

---

1. The declaring class is omitted in the methods’ descriptor for brevity. The two methods `toString` in Fig. 5 are declared in two different classes.
Steamloom. The entry point of the module, the method `weaveIn`, was given to the participants and they were asked to investigate refactoring possibilities that would improve the structure of the module. For this purpose, an understanding of the internal module structure was needed.

**Observations.** The participants tried to understand the module’s control flow by exploring the calls relationship starting with the `weaveIn` method. All tools provided support for this task.

Participants P3 to P5 (using Creole and SEXTANT) produced a control flow graph with a few exploration steps. The user of Creole filtered all relationships except for the calls relationship. The layout algorithms automatically created a representation which was rated as very helpful. The users of SEXTANT constructed the call graph by exploring the call relationship recursively. Thereby, they adapted the layout manually so that called methods were located below their callers. The call graphs were then used 1) as a kind of an overview of the current working set and 2) to find patterns that indicate methods to refactor.

For instance, Fig. 6 shows the exploration graph of P3 using SEXTANT. A first example of a visual pattern that indicates a potential “bad smell” is shown in the diamond-like structure at index 1. The structure indicates that the bottommost method of the diamond (`removeNOP`) has no outgoing calls to other methods, i.e., it is used as a “service provider” which does not further contribute to the control flow. The removal of NOP operations should be performed after each weaving process, independent of the kind of code generated. Thus, the call to `removeNOP` can actually be moved upward in the call flow graph into the method `weaveIn`. The visualized call graph refactoring was instructive in revealing this refactoring possibility. Another example of a visual “bad smell” is shown in index 2: a method with only one incoming and one outgoing arrow. This pattern often reveals methods that forward to the called method, thereby passing a default value; such methods can often be in-lined.

The users of JQuery experienced difficulties in understanding the control flow due to lack of referential integrity of the tree-based representation of query results. As they explored the calls relationship, methods appeared several times in the view. Before exploring a certain method further, users often searched the tree to ensure that it was not previously explored. Moreover, the representation did not support finding refactoring opportunities. Instead, the source code was investigated.

After identifying refactoring candidates, the participants looked at the source code to test their hypothesis. JQuery and SEXTANT use the Eclipse editor while Creole visualizes source code within method nodes. To see the source code of different methods, participants using Creole had to change the view by zooming in and out frequently. Zooming into a particular method resulted in hiding other methods. Further, to change the code, they had to open the Eclipse editor and scroll to the corresponding method manually.

**Results.** For the comprehension process in this study, the visual representation of the explored control flow (R3) was crucial for two purposes: 1) to create and retain the mental map and 2) to reason about the explored elements and their relations. To facilitate the construction of the mental map, it is important that the visual representation matches the graph-based structure of a control flow. To reason about an element, it is important that all of its relations are visualized explicitly and referential integrity is maintained. Thus, for both purposes, a graph-based representation was appropriate, while a tree-based one was counterproductive.

The study also illustrates the importance of traceability (R5). Although viewing the source code was supported by all tools, only JQuery and SEXTANT enabled switching seamlessly between the graphical representation and the code. Both tools enable seeing both the graphical representation and the low-level implementation details. Thus, the developers can see software elements in their context at two levels of abstraction simultaneously. Creole’s approach of visualizing code inside the graphical nodes was rated as unsuitable. First, zooming into a corresponding method changes the visual representation and causes a loss of context. Second, for the refactoring, the developer had to navigate to the corresponding source code location in the editor, which introduced some accidental complexity.
5.4 Concern Exploration

Task. The subject of the third case study was an object-oriented telecommunication application with two cross-cutting concerns [45]. The task was to expand a clue of the timing concern, i.e., domain knowledge about the existence of timers. This involves finding all relevant program elements and relationships of the concern scattered throughout the system. The participants were asked to describe the structure of the concern and their rationale for classifying each particular element as part of it.

Observations. All participants started by searching for the Timer class and continued to explore elements related to it. The users of Creole did not accomplish the task due to difficulties in following specific relations. For instance, to follow a method call, they had to zoom out of the caller, follow the arc representing the call relationship, and then zoom into the class of the callee to see the actual called method. After a few navigation steps, Creole’s users did not know anymore which elements were already explored. This was more difficult since they could not hide elements irrelevant for the timing concern. Creole visualizes all elements of a given subsystem at once, but the participants were interested in only a small subset of elements and relations.

Participants using JQuery and SEXTANT navigated along different relations such as calls or accesses field. The source code of the elements revealed in such exploration steps was investigated in most cases; sometimes the relevance was judged by the element’s name only. The participants used this schema recursively until they found all elements of the concern.

However, as in the previous case study, the user of JQuery had to check whether a newly revealed element was already explored by searching for it in the tree. Further, it was not possible to hide single elements; only whole subtrees could be removed. Finally, it was difficult to reason about the concern’s structure because relevant elements could appear several times in the exploration tree. The developer had to search the whole tree to find all elements that could be part of the concern. Furthermore, relationships between different elements were explicitly visible. The participants used the graphical representation to reason about the concern’s structure.

Results. The study shows that both searching and browsing are needed (R1). Developers started top-down using their knowledge about the timing concern to find an initial element. Then, they switched to bottom-up comprehension by exploring the neighborhood of this element.

The explicit representation of the exploration results was crucial for reasoning about the concern’s structure (R3). If referential integrity is not maintained, one has to find all occurrences of the element to reason about and combine their relationships manually. Elements and relations that are irrelevant for the task at hand clutter the display and complicate the comprehension process.

Finally, the study also illustrates the need to complement the graphical representation with the source code (R5). The former was used to understand the structure of the concern, while the latter was used to decide whether an element is part of the concern. Hence, both views need to be visible simultaneously so that low-level information can be interpreted in the context of the “big picture.”

5.5 Verifying a Hypothesis

Task. The fourth study covered another understanding task in Steamloom during a perfective maintenance task. Steamloom’s weaving process generates new code by using the bytecode toolkit BAT [46], which represents methods by bytecode as linked lists of instruction objects. Instantiation of such objects should occur on two occasions only: during class loading and during weaving.

Class loading is handled by BAT and weaving is handled by Steamloom. Hence, BAT bytecode instruction objects should be exclusively instantiated in the Steamloom weaving module and in BAT. In particular, the underlying VM source code should be kept clean of such instantiations for maintainability. Developers were told that BAT and Steamloom are encapsulated in two packages, de.tud.bat and de.tud.aorta.steamloom, respectively. They should report all locations where instructions are created and they should analyze whether the localization restriction is obeyed.

Observations. All participants followed the same procedure: They searched for Instruction instances and then they tried to explore all of their instantiations and aggregate those to modules.

The Creole participants got lost during this task. They navigated from the Instruction interface to all of its subtypes. Then, they tried to understand where instructions of the concrete subtypes are created. This was difficult for three reasons.

First, numerous instructions had to be investigated and the developers had to synthesize the results manually. Since Creole always provides a full overview, it was not possible to expand only a subset of interesting elements of a module. Although Creole enables focusing on specific parts of the software by zooming and filtering elements and relations according to their type, these means were not sufficient for the task at hand. Second, without means for aggregating all instruction subtypes or for defining a query that presents all creators of a type and its subtypes, the view became cluttered and the developer got lost. Third, to follow the creation relationships that span two subsystems, many zoom out and zoom in actions were needed.
The other participants (P1, P2, and P4) also had to consider the entire *Instruction* type hierarchy. JQuery and SEXTANT did not provide "built-in" queries for retrieving constructor caller sites that also take the inheritance hierarchy into account. However, P2 and P4 used the extensibility of the respective tools to write a new query for this purpose. For this purpose, they investigated the queries of existing relationships, e.g., for the instantiates and subtypes relationships, and wrote a new, similar query by adapting them. We did not observe problems in defining the query.

The query was then used during the exploration. The developers navigated from the *Instruction* interface to all creation sites of any instruction subtype. Using JQuery, the resulting exploration tree instantly showed that the concern was localized because the revealed methods were organized along their containment hierarchy (package, class, member) in the tree. The user of SEXTANT used a query on the method nodes creating *Instruction* instances to determine their declaring classes.

The final exploration graph constructed by the SEXTANT user is depicted in Fig. 8. Due to the explicit representation, it is obvious that nearly all methods creating BAT instructions (the nodes in the center) were methods of Steamloom’s *CodeGen* class (the rightmost node). Only one method is a member of another class, namely, *VM_AspectUnitRegistry* (the node at the upper right corner), an internal Steamloom class.

The second developer using JQuery (P1) did not write a dedicated query, but tried to navigate from the *Instruction* interface to all of its subtypes and then explored the creators of each subtype. Even though he could finish the task, it was much more difficult and time-consuming to check whether the concern is localized.

**Results.** This study showed the need for integrated comprehension across different levels of abstraction (R1). First, one needs to retrieve those program elements that create objects of a particular type. These creation sites then need to be aggregated to higher-level modules, in order to see whether the concern is localized.

Due to the lack of means to filter specific relations in Creole, it was not possible to aggregate the creation sites of instructions to their declaring classes. JQuery and SEXTANT enabled aggregation of the methods creating *Instruction* objects with respect to their declaring classes. However, participants complained about the lack of support for defining new aggregations beyond those present in the software structure. For instance, it was not possible to aggregate all elements that belong to Steamloom. We will elaborate on this in Section 7.

Another interesting result is that the extensibility of JQuery and SEXTANT significantly facilitated the comprehension task (R4). By defining a new query suitable for the task at hand, the exploration process was shortened and the localization of the instruction creation concern was clearly visible in the resulting exploration tree or graph, respectively. However, the study does not provide any evidence on the ease of writing queries from scratch, because developers could adapt existing queries.

### 5.6 Synthesis and Threats to Validity

The four case studies, each covering a subset of the five requirements, taken together provide initial evidence for each of the requirements, R1 to R5. We have shown that comprehension tools supporting all five requirements such as SEXTANT are more effective in gaining understanding during various comprehension tasks. Tools that support a subset of the features may support the developers in some comprehension tasks, but the creation of a mental map was hindered in other cases.

Several actions were taken to mitigate threats to the validity of the case studies. To minimize the divergence between the measures of the participants’ behavior and their real behavior, transcripts of their actions were minuted. To minimize the investigator bias in choosing the maintenance tasks, we created the studies based on questions that arose during the maintenance of the software systems. Further, each participant used all three tools to reduce the impact of developers’ experience on the results. Finally, to avoid inconsistencies, only a single experimenter conducted the studies.

The design of the experimental evaluation also has properties that limit the generalization of our findings. First, we observed only five developers that all had good programming experience and a high level of proficiency with Java and the Eclipse IDE. Second, we investigated only four maintenance tasks that required a partial understanding of the software. To gain additional evidence, a broader study with more tasks and more participants with different backgrounds needs to be conducted in future work. For other tasks that require understanding very large parts or the whole system, e.g., software architecture reconstruction, tools like Creole that show the whole system at once are more appropriate than SEXTANT. Yet, the focus of this paper was on exploration support rather than on support for creating overviews of software systems.

### 6 Comparison to Other Requirements

Other researchers have imposed other requirements on software comprehension tools focusing on cognitive aspects [17], [22] and on functional and nonfunctional properties [23]. We argue that by satisfying R1 to R5, many of these requirements are also met by evaluating SEXTANT against them.

#### 6.1 Tool-Specific Cognitive Design Elements

Storey et al. [22] propose a hierarchy of cognitive design elements consisting of two main branches, namely the improvement of program comprehension and the reduction...
of cognitive overhead. They derive 15 requirements from them (E1 through E15). In the following, we list properties of SEXTANT and refer to the requirements they fulfill.

The first branch concerning improvement of program comprehension can be further categorized in enhancements to bottom-up, top-down, and integrated comprehension. SEXTANT contributes to all three categories, as elaborated in the following.

Bottom-up comprehension. According to Storey et al., an exploration tool should support the three main activities involved in the bottom-up comprehension process: 1) the identification of software elements and their relations, 2) browsing code in delocalized plans, and 3) chunking low-level information to higher-level abstractions.

SEXTANT comes with an extensible set of elements and relationships. All elements of a software are represented as nodes, while relationships are depicted as edges in the exploration graph. Thus, we provide immediate and visible access to the explored software elements and the syntactic and semantic relations between them (E1).

The effect of delocalized elements is reduced because the exploration graph combines all relevant elements and relations independently of their location (E2). The traceability property ensures that users can switch from the graphical view to the source code to see the software elements in their context.

In the current version, SEXTANT provides very limited abstraction mechanisms (E3) by enabling the linking of two elements that are related in a parent-child manner and the continuing exploration with the parent element. For instance, consider a Java system in which modules map to the package structure. In such a scenario, developers can navigate from a type to its declaring package (representing a higher-level abstraction) and continue the exploration with the latter element. However, we do not provide means to create user-defined abstractions and visual aggregation can only be achieved by hiding the low-level elements manually. This issue is further discussed in Section 7.

Top-down comprehension. Top-down comprehension is enhanced by facilities to verify hypotheses quickly (E4). SEXTANT enables to find clues to verify or reject a hypothesis by means of various search queries. However, there are currently no means for expressing hypotheses explicitly.

It has been a well-thought design decision not to support predefined overviews of the system architecture at various levels of abstraction (E5) in SEXTANT. Rather, the developer can access and explore software elements at a high level of abstraction and use different queries to gain access to elements at a lower level of abstraction. Thus, various levels of abstraction are fused together into a single view.

Integrated comprehension. Supporting integrated comprehension means that developers are able to construct “several linked views representing a variety of cross-referenced mental models.” Although SEXTANT has a single graph-based view, it is an integrated view and enables exploration of several different kinds of relationships. Thus, we support multiple logical views, but those are fused together into one graphical representation.

This has two implications: First, the construction of multiple mental models is supported (E6). Second, the different logical views are automatically cross-referenced, which facilitates the construction of the mental model (E7). The latter two requirements are supported on the program level but not, e.g., on the domain level.

The second branch of cognitive design elements comprises those aiming at a reduction of the user's cognitive overhead. Cognitive overhead can be alleviated by providing good navigation facilities and meaningful orientation clues and by effectively presenting the information so that it can contribute to software comprehension.

Navigation support. SEXTANT provides directional navigation by enabling users to browse arbitrary kinds of relationships (E8). It also supports “arbitrary navigation,” i.e., the ability to navigate to locations not necessarily reachable by following an application or user-defined link (E9). A user can navigate to elements directly by means of SEXTANT’s search feature. Further, one can use, e.g., Java annotations to mark elements of interest and use those annotations to reaccess the elements in the future. By visualizing all steps during the exploration as linked elements within a single integrated view, navigation between the various mental models (i.e., logical views) is also possible (E10).

Orientation clues. Orientation clues are provided by means of the exploration graph itself. In SEXTANT, the element that is currently in the developer’s focus is highlighted graphically (E11). Further, the graph communicates a map of the exploration process. Thus, the path that led to the focus is always shown explicitly (E12). Additionally, each element which is part of the exploration graph can be further explored (E13). By right-clicking on any node, the tool indicates all relations that can be explored for this type of element.

Reduction of disorientation. The last two cognitive design elements concern the reduction of disorientation.

First, additional effort for user-interface adjustments should be minimal (E14). In particular, disorientation is often caused by switching different views. Because SEXTANT makes use of a single, integrated view, there is no need to switch to different specific views.

Second, the presentation style should be effective, i.e., display the information in a more meaningful manner (E15). SEXTANT currently provides two different layout styles and the appearance of nodes and edges can be customized. This can be used to communicate specific characteristics, e.g., a hierarchical nature using a tree layout.

Another important property of SEXTANT is that only those elements that are of interest, i.e., that are explored by the user, are visualized in the exploration graph. This filtering diminishes the need for more effective presentation styles to cope with scaling issues.

However, exploring larger concerns also leads to larger exploration graphs, inducing the need for more effective visualization techniques. We will discuss this issue in Section 7.

6.2 General Cognitive Design Requirements

Zayour and Lethbridge [23] investigated the low adoption problem of reverse engineering tools. They applied cognitive analysis and derived two categories of general cognitive design requirements that should be satisfied by maintenance tools.

Short-term memory utilization. Requirements that facilitate a minimization of the number of artifacts that have to be kept in short-term memory (STM) comprise the first category. Three subrequirements are derived.

First, the required artifacts should be easily available (A1). This is supported by the extensibility of SEXTANT. Due to the usage of Magellan as its underlying framework, developers can integrate various kinds of artifacts and
define relations between different kinds of artifacts. Such relations can then be explored independently of the kind of artifact.

Second, new information should be linked in meaningful ways with existing information (A2). This is automatically achieved during the exploration by making the semantic relationships between elements explicit in terms of edges in the exploration graph. Further, the referential integrity of our view also ensures that relations between newly revealed elements and already existing elements are visualized explicitly; thus, new information is shown in the context of already explored elements.

Third, uncertainty during exploration should be reduced (A3). Uncertainty is an intrinsic property of every exploration. However, having an explicit exploration path enables users to backtrack into an alternative exploration path without getting lost.

Short-term memory fading. The second category of requirements is the minimization of STM fading, which can be further split into two subrequirements. SEXTANT contributes to both of them.

STM fading can be reduced by minimizing the time that artifacts have to be retained in the STM (B1). A major portion of time is spent acquiring related elements successively. Our tool provides means to specify user-defined queries that enable users to browse to a related element with a corresponding kind of relationship in a single step. Moreover, all explored elements are visualized explicitly in the graph. This frees users from holding all information in STM; instead, they can reconstruct the knowledge quickly from the exploration graph.

The number and complexity of intermediate steps in acquiring the next artifact should also be minimized (B2). In SEXTANT, developers select a kind of relation from the context menu of an element and the tool visualizes all related elements with respect to the given relation. Further, user-defined queries can automate several steps of a manual exploration, so that users can explore the information within a single step.

### 6.3 Functional and Nonfunctional Requirements

Singer et al. [17] created a list of key requirements for a software exploration tool consisting of three functional and seven nonfunctional requirements. The major difference to our work is that they focus only on those requirements matching the current workplace activities in order to facilitate tool adoption. The developers that participated in the study mainly used the search feature; hence, the requirements discussed in [17] predominantly concern searching. Further, our requirements are derived from the information demand of developers as well as based on empirical studies of programmers and focus on the broader field of software exploration.

**Functional requirements.** First, search capabilities are an essential part of navigation (F1). The observation showed that means are needed to search by exact name, by way of pattern-matching as well as for “semantically significant” information, e.g., variable names or method calls. All these types of searches are possible in SEXTANT by means of queries.

The second functional requirement calls for displaying all relevant attributes of elements and the relationships between them (F2). All relationships of a given element can be made explicit in SEXTANT by exploring the corresponding queries. This can also be used to show the attributes of an element. For instance, navigating from a class to its declaring package makes the latter explicit. However, it is also possible to customize the appearance of each element type. For instance, developers could use a UML-like visualization of types to display all declaring methods and fields.

Third, a persistent search history is requested (F3). This feature is not yet available in SEXTANT. We will discuss a possible extension of our tool to persist exploration graphs in Section 7.

**Nonfunctional requirements.** The first nonfunctional requirement addresses scalability, demanding that tools should be able to process several million lines of code (NF1). At the same time, most queries should respond without perceptible delay (NF2). As we will discuss in detail later, SEXTANT can handle large software (depending on the available main memory) and most queries execute in imperceptible time.

Backed by the usage of Magellan as the data layer and the abstract software model in the form of graphs, our tool is capable of processing source code written in various programming languages (NF3). Developers can integrate artifacts for a given programming language using a corresponding XML representation (e.g., [47], [48], [49]) and define queries across these kinds of artifacts in a uniform manner.

Further, interoperability with other tools is required (NF4). Exploration tools should be well integrated and incorporate all frequently used facilities that developers are used to (NF6). Due to SEXTANT’s tight integration in the Eclipse IDE, developers can use all features the environment provides. The architecture of our tool stack enables tools to access the underlying Magellan framework so that queries can be reused in different applications.

A kind of pluggable user interface is also required (NF5). Currently, our tool exhibits features that enable developers to exchange the visualizations for a given kind of element or relationship. Further ideas on a more flexible visualization are discussed in the following section.

Finally, tools should present the user with complete information in a manner that facilitates just-in-time comprehension (NF7). SEXTANT’s extensibility with respect to queries enables browsing to latent information. For instance, a developer can see if a method of an EJB is executed in a transactional context.

### 6.4 Summary

The comparison of SEXTANT to the three lists of requirements for general comprehension tools showed that our tool satisfies most of them. SEXTANT provides support for bottom-up, top-down, and integrated comprehension. The query-based approach facilitates navigation, and the explicit exploration graph using a single integrated view avoids getting lost.

By following an on-demand exploration approach instead of providing views that show all elements of a software, we reduce the short-term memory load and minimize fading.

Finally, SEXTANT also supports most functional and nonfunctional requirements identified by studying developers’ work practices.

### 7 Discussion and Future Work

The experience with SEXTANT has revealed several open issues to be targeted in future work.
7.1 Scalability
To use software exploration tools in an industrial context, they should scale up to large-sized projects with several million lines of code and respond without perceptible delay [17]. SEXTANT is a research prototype and does not scale up to very large projects because all data is held in main memory. However, the case studies with Steamloom provided promising first indications about SEXTANT’s performance in terms of memory and computation time.

Steamloom consists of roughly 200,000 lines of Java code, about 150,000 of which belong to the underlying Jikes RVM and another 30,000 belong to the bytecode toolkit BAT. The underlying database produced by Magellan had a size of about 50 MB when made persistent. However, during the usage of SEXTANT, the whole program database was kept in memory and required about four times that space. All queries were, despite the large memory requirements, evaluated in about one second at most, using a dual 3.06 GHz PC with 2 GB main memory. We have not conducted detailed performance measurements, but the speed experienced during working on Steamloom was satisfactory in all cases.

The performance of SEXTANT is mainly driven by Magellan’s performance. We are currently working on an XML wrapper around the object representation of the source code produced by BAT. Using this technique, a subtree of the XML database is loaded into main memory on demand only. This is particularly suited to exploration tools because query results during an exploration task are typically a very small subset of the whole software space.

First experiments show that we can reduce the memory requirements significantly while obtaining the possibility to use standard Java XML libraries. Further, we observe that the runtime of complex queries often can be improved significantly by a refactoring of the query or by using an XQuery processor with support for indexes [50]. We hope to be able to use SEXTANT on industrial-sized systems in the near future.

7.2 Extensibility
Currently, SEXTANT provides means to explore Java software only. However, it is extensible with respect to 1) the kind of software artifacts and 2) the software elements, the relations one can navigate along, and the queries that can be used for searching.

To integrate a new kind of document, e.g., another programming language to enable cross-language exploration, a mapping between the document’s content and a corresponding XML representation is needed. Developing such a mapping from scratch is not an easy task, but a body of work already exists (e.g., [47], [48], [49]). Other development artifacts, such as configuration files or UML diagrams, often are XML representations per se (e.g., [31], [51]).

To extend the navigation facilities of SEXTANT, developers have to write corresponding queries. We took three measures to reduce the effort to write new queries. First, we decided to use the standard query language XQuery in favor of designing a specialized one. Second, we provide a broad library of commonly used functions that can be used in user-defined queries. For instance, functions exist to select only a project’s sources (filtering out libraries) or to get all subtypes of a given Java type. Third, all queries provided by SEXTANT are well-documented and can be used as examples.

We observed that developers who are familiar with XQuery-like languages could start writing queries instantly given a schema documentation and query examples. However, developers not familiar with query languages needed a significant amount of time to learn it. A group of students that implemented metrics for SEXTANT was able to write working queries from scratch after a one-day introduction to XQuery. However, the resulting queries were implemented in a procedural way and, thus, hard to read.

7.3 Storing Explorations
Software exploration aims at gaining an understanding of the structure or behavior of a software system. Typically, this knowledge is used in successive code modification tasks, but it can also be used to improve collaboration, i.e., by forming a group memory or by recording the gained knowledge for future use [52]. To enable this knowledge exchange, means are needed to make the exploration graph persistent.

One can store the information as a first-class concern [34] or in an extended concern graph that is not restricted to a fixed set of kinds of elements and relationships [39]. Such a feature would also serve to keep track of problem-solving sessions (requirement F3 in [17]). It is not clear, though, whether only the structure should be stored or also the visual properties so that the representation always looks the same and “temporal continuity” of the view is maintained [53]. Another question is whether the exploration should be stored intensively (i.e., the queries used in the navigation steps are made persistent), extensionally (by storing the elements and relations), or both.

7.4 Flexible Visualization
Although the visual representation of query results as a graph has proved helpful in the case studies, two limitations have been observed. First, it does not scale: The larger the graph, the less comprehensible it is. Second, no single visualization is appropriate for all tasks. The success of a representation depends on 1) whether it makes particular information the user needs accessible and 2) how well it copes with the varying information requirements the user has in the process of performing various tasks [54], [55].

To address these issues, we are working on a more flexible visualization layer. Currently SEXTANT users can only exchange the renderer for a given element or relationship type as well as the graph layout algorithm. However, to facilitate the construction of a mental map, the user should be able to customize the graphical notation so that it matches the structure of the problem [56].

We plan to provide means for letting developers choose 1) the basic visualization technique, such as graphs or tree maps, 2) layout mechanisms, and 3) the appearance of nodes and relationships. Furthermore, developers should be able to filter and aggregate nodes along various properties, e.g., the package name of a class. The latter mechanism could also be used to enable user-defined abstraction mechanisms (E3 in [22]). A more detailed description of our work on flexible visualization can be found in [57].

7.5 Improving Comprehension Support
To generalize our findings, we want to evaluate SEXTANT in the context of other software maintenance projects. Besides evaluation purposes, the goal will also be to collect more experience with common tasks in such projects and their needs on search and browsing capabilities. First, this knowledge will serve to extend SEXTANT’s set of predefined queries. Second, we plan to extend the set of
supported comprehension tasks. The tasks in the case study all had an exploratory nature: The developer first has to know or find an initial program element to start the exploration from. This program element is then used to explore its context.

For other reverse engineering tasks, e.g., the identification of the building blocks of a software, such an initial program element is not known and hard to find. Rather, predefined views that, for instance, visualize the packages and their dependencies are more appropriate. We think that the query-based approach of SEXTANT can also be used to produce such views. Their integration into an exploration tool seems beneficial because, in that case, once developers focus on a single module, they can directly continue the exploration of its context.

8 Related Work

Many tools have been developed to support the understanding of software systems. One category of software comprehension tools focuses on visualization facilities (e.g., [7], [8], [9], [10], [11]).

Two well-known examples are Rigi [9] and SHriMP/Creole [10], [42], which has been described in Section 5. Rigi is a system for reverse engineering, primarily capable of the identification of subsystems by certain criteria, e.g., file containment or element names. The results of the identification are then visualized. All subsystems form a hierarchy, which is displayed in an overview window, but the details of a subsystem, i.e., the contained elements, are represented in their own window. Thus, Rigi follows a multiwindow approach. SHriMP and Creole, on the other hand, use a single view with well-known visualization techniques such as fish-eye view or pan and zoom.

In contrast to SEXTANT with its lightweight visualization, both tools provide more complex visualization techniques. However, recent studies revealed that developers are often swamped by too many elements [11], [58] and too complex visualizations [24]. For example, the existence of multiple, nonintegrated views can cause disorientation in the case of Rigi, whereas the SHriMP visualization can result in an information overload. Our approach differs in that not the whole system is visualized, but the developer explores software elements of interest step by step. Even though Rigi and SHriMP provide support for source code navigation using hyperlinks and for context navigation, their browsing capabilities are limited due to a small number of queries and the absence of means to add new queries or customize existing ones. SEXTANT provides means to add new relations and the tight Eclipse integration enables developers to switch to the source code seamlessly.

Other tools aim at combining the two navigational styles searching and browsing, a prerequisite to supporting the integrated model of software comprehension. Examples of those tools are Hy+[59], Ciao [12], The Searchable Bookshelf [16], and SPOOL [15]. All these tools have in common that they are based on a kind of repository, e.g., a fact base or a database, and provide advanced query mechanisms allowing a developer to extend the tool by defining new queries. Although this enables searching or browsing using diverse relationships, one cannot fuse different views. If a developer uses, for instance, SPOOL and starts an exploration with a query, it is common that the next step will be the further evaluation of the results. Even though it is possible to use the results of the former query as starting points for a new query, one looses the exploration path, which is essential to building up a mental map of the software system [20].

On the contrary, SEXTANT presents the results of a complete exploration in a single view. This results in an explicit representation of the exploration path, preventing developers from getting lost during the exploration.

Feat [39] is a tool to create and manipulate representations of concerns. Developers can navigate along different semantic relationships between program elements and add elements of interest to a concern. All concern elements and their interrelations are abstracted in a concern graph representation.

SEXTANT’s capabilities to search, browse, and visualize program elements are more advanced when compared toFeat with its fixed program model. We provide means to search and browse using different semantic relations and in different kinds of artifacts. Furthermore, our graph-based visualization is more appropriate for understanding interrelations between program elements than a tree-based one. However, the main contribution ofFeat is a means to make concern descriptions explicit using concern graphs. We will investigate the use of those representations to persist results of an exploration.

The tool most similar to SEXTANT is JQuery [14] (see Section 5). In contrast to JQuery’s tree-based visualization, SEXTANT uses a graph-based representation which improves the comprehension of the relationships between the system’s elements. Each program element occurs at most once in the view. If we discover a new element and there are relations to other elements in the view, they get automatically visualized so that each relation is made explicitly visible. Furthermore, we use Magellan as our data layer. This enables to store and query different kinds of artifacts in a uniform way. Those cross-artifact query capabilities broaden the scope of possible applications and enable developers to write domain or technology specific queries, even if source code written in a language other than Java is used.

A different approach of software exploration, namely, backpacking exploration, is described by Favre [60]. The work is similar to SEXTANT by providing means to explore different kinds of artifacts. While SEXTANT uses an XML database as the underlaying source for queries, the GSEE backpacking framework provides a generic successor interface with a single method returning all related elements for a given one. This enables the integration of various information sources. For instance, one can use the interface to integrate an object-oriented database, or one could create an implementation of it using Java introspection to find related elements for a Java class. The simplicity of this interface facilitates the usage of existing libraries as new kinds of information sources with almost no preparation efforts. Hence, the metamodel can be elaborated during the actual exploration by integrating new source components interactively, which promotes the discovery of new concepts on the fly.

While one needs to create a transformer to an XML representation for new kinds of artifacts in SEXTANT, a software backpacker adds different extractor tools to integrate a new kind of artifact on the fly. Thus, the backpacker approach is particularly suitable for exploring new “software landscapes” with different, noncommon kinds of artifacts. However, SEXTANT is more appropriate in the case of common artifact kinds because the query facilities enable defining relationships over various artifacts
in one step instead of having to explore those relationships incrementally.

9 SUMMARY

In this paper, we presented five functional requirements for software exploration tools and provided initial evidence that these features are needed to effectively assist developers in understanding software. As a proof-of-concept, we presented SEXTANT, a tool with support for all five requirements.

Individual requirements are also supported by other comprehension tools but, as discussed in the related work section, to the best of our knowledge, none of the related tools supports the combination of features provided by SEXTANT. By four case studies, we have shown that tools which support the requirements posed in this paper ease the comprehension process compared to tools which support only a subset of the requirements. Finally, we have also shown that SEXTANT satisfies most requirements posed by other researchers.

ACKNOWLEDGMENTS

This work has been partly funded by the AOSD-Europe Network of Excellence, EU grant no. FP6-2003-IST-2-004349. The authors would like to thank S. Dukanović and M. Vekić for implementing parts of SEXTANT. The authors also thank I. Aracić, C. Bockisch, and the anonymous reviewers for their helpful comments.

REFERENCES


For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.