Comprehensive Software Understanding with SEXTANT

Michael Eichberg  Michael Haupt  Mira Mezini  Thorsten Schäfer
Software Technology Group
Department of Computer Science
Darmstadt University of Technology
{eichberg,haupt,mezini,schaefer}@informatik.tu-darmstadt.de

Abstract

Current tools for software understanding mostly concentrate on one comprehension technique, e.g., visualization, or bottom-up navigation through software elements via hyperlinks. In this paper, we argue that to effectively assist developers in understanding today’s software systems, a combination of several comprehension techniques is needed including seamless integration of top-down querying and bottom-up navigation strategies that work across different kinds of software artifacts; furthermore, application-domain and/or technology specific relationships between software elements should be taken into consideration; last but not least, a tight integration of such tools into development environments is crucial. We present SEXTANT, a software exploration tool tightly integrated into the Eclipse IDE that satisfies these requirements. In two case studies, we demonstrate how SEXTANT’s features are conducive in tracking down the source of erroneous behavior, respectively, in discovering ‘bad smells’ in the software structure which should lead to code refactorings.

1 Introduction

To maintain and/or extend software systems developers need to understand their internal structure to ensure that changes do not break the intended behavior of the system [26]. Unfortunately, appropriate documentation of the system structure is often missing or outdated; even when accurate documentation is available, tool support for software comprehension is indispensable, given the complexity of today’s software systems.

Tools for software comprehension can be classified in two groups. On the one hand, there are software visualization tools providing sophisticated visualization techniques for a software system [6, 11, 15, 23, 24], e.g., CodeCrawler [6], or SHriMP [23]. On the other hand, there are software exploration tools that provide means to navigate along a software system [5, 10, 17, 19, 20], such as tksee [20] or JQuery [10].

Recent studies indicate that the navigational aspects are most important in a software comprehension tool [1]. Developers search for elements in a software system over and over again and navigate through their relations. During this exploration process, also called Just in Time Comprehension [20], the developer constructs a mental map of the visualized information [7]. Thus, software exploration tools seem to be most appropriate for supporting software comprehension and are, for this reason, in the focus of this paper. We pose and justify five requirements on such tools, labeled R1 to R5 below, and present a software exploration tool that fulfills them.

R1: Integrated comprehension. Three basic software comprehension strategies have been described [25]: Bottom-up, top-down, and mixed strategies. Developers using a bottom-up approach [18] achieve a high-level software comprehension by starting to read the low-level source code and stepwise abstracting from it. Software engineers following the top-down approach [4] use their general domain knowledge to formulate an initial hypothesis about the software. The initial model is then refined by trying to verify it and by searching for corresponding structures in the code. Mixed strategies assume that developers are capable of using both aforementioned strategies [13, 21]. Van Mayrhofer et al. [27] 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. Following on this work, we require that software comprehension tools should support an integrated comprehension subsuming both, the bottom-up and top-down approach.

R2: Cross-artifact support. Modern tools, such as persistence frameworks, and component technologies, such as Enterprise Java Beans (EJB), aimed at better mastering the software complexity, ironically also cause a new kind of complexity to emerge. The developer using such technologies is forced to work with a multitude of different kinds of
artifacts: Besides source code, a large number of external libraries is often used and information that affects the runtime behavior is stored in XML or properties files. As a consequence, it is no longer possible to analyze and comprehend a software project without considering and aggregating information contained in all kinds of artifacts. Hence, exploration tools are required that enable software comprehension across artifact borders.

R3: Explicit representation. To support the creation of a mental map of the software all of its elements and the relations between those should be visualized explicitly and the referential integrity among them should be maintained as the navigation process unfolds. Unfortunately, this is not the case in many mainstream IDEs. For instance, in Eclipse it is possible to use hyperlinks to get from one software element to another: One can navigate from a class to all its methods and from each method to all of its callees, and so on. The path followed by such an exploration is, however, not visible, making it hard to build the mental map [10]. By using specialized views such as the call hierarchy view, it is possible to see the path for a single kind of relationship. But, switching to a different kind of relationship requires switching the view and thus can cause disorientation [22]. Furthermore, the navigation often lacks referential integrity in the sense that the same element may appear several times potentially in different views of the IDE. For instance, assume a method m1 is called by two other methods m2 and m3. In Eclipse, one can use hyperlinks to navigate from m2 and m3 to m1. However, in this process m1 will appear twice in two apparently unrelated views—the list of methods called by m2, respectively by m3. The referential integrity of m1 is not maintained during the exploration and it is hard to discover that m2 and m3 are related by the property of calling m1.

R4: Extensibility. Software exploration often involves navigating to common software elements and along common relations. For instance, common elements one would like to navigate to while exploring a Java application include classes, methods, and fields; common relations frequently used to navigate along are inheritance and call relationships. In addition to such common elements and relations, specific application domains or specific libraries in use may require to navigate to new kinds of elements and along new kinds of relationships. In an EJB project, e.g., technology specific relations between a class, its corresponding public interfaces and related elements in deployment descriptors become relevant relations to navigate along. However, tool developers cannot foresee all contexts in which an exploration tool is used. Hence, we conclude that software exploration tools should be extensible to accommodate for domain-specific navigation elements and relationships as needed.

R5: Traceability. In most cases, comprehension is not an independent software development task. Rather, we use exploration to understand a system to continue with a modification, which is done at the code level. Hence, the ability to switch instantly and seamlessly between the graphical notation and the corresponding source code is essential for practical use [1]. Furthermore, often the exploration process might only give us hints; the actual understanding or the verification of our hypotheses may require switching to the source code representation of the system.

The discussion so far makes clear that software exploration tools that support these five requirements are most appropriate for comprehension processes. Unfortunately, as we will discuss in Section 4 current tools only provide support for a subset of the features. This has been our motivation for developing Sextant, a software exploration tool with support for all five requirements, which we present in this paper.

Sextant follows a query-based approach for both searching program elements and browsing along different kinds of relations. These two navigational styles support bottom-up and top-down comprehension. The integrated comprehension model is enabled by an integrated view: The elements discovered in any search or browsing activity are visualized as nodes in a graph, while relationships between elements are depicted as edges of the graph, whereby, all elements and relationships are explicitly represented. By building upon Magellan [8], Sextant offers functionality for navigating across various kinds of artifacts involved in a modern software project. Furthermore, Sextant is extensible with new kinds of artifacts, new node types, and user-defined relations. By being tightly integrated into the Eclipse IDE Sextant provides means to seamlessly switch between the graphical notation and the source code representation.

In two case studies, we demonstrate how the tool was used to understand the source of a bug, respectively to discover overly complex structures which led to refactorings. The case studies also show that Sextant exhibits acceptable performance even for large software systems.

The reminder of this paper is organized as follows. In Section 2, the architecture of Sextant is presented. Section 3 describes the application of Sextant in two case studies. Section 4 discusses related work. Section 5 summarizes the paper and presents areas of future work.

2 Sextant’s Architecture

Sextant has a three-tier architecture shown in Figure 1, comparable to the proposal described by Lanza [12]. The data layer provides the raw data and means to extract information. This layer is an XML database which can be queried using XQuery [2]. The integration layer processes the extracted information such that it is conform to the
meta-model of the data layer, i.e., based on the results of the executed queries, the integration layer constructs a graph. The latter is visualized by the visualization layer.

Figure 1. Architecture overview of SEXTANT

2.1 Data Layer

The data layer stores all project artifacts in a database and provides access to the database by means of queries. It is based on Magellan [8], a cross-artifact information engineering platform. All artifacts of a software are transformed into a uniform XML representation, which automatically gets updated in the case of changes. This enables to define queries across different kinds of artifacts using XQuery [2].

For the purpose of this work, we have extended Magellan to support three different types of queries. First, there are queries that search the entire database, e.g., to find a type definition, or a method. These queries return references to the nodes of the XML database that match the selection criteria and are used for the search feature. Second, there are queries that are defined with respect to a previously selected node, i.e., queries that need a specific context to be evaluated. For example, given a query of the first type which returned a class declaration node, we can set this node as the context for a query to get all sub- or super-types. In order to enable to browse through a software project these queries are of utmost importance. Third, there are queries that, given a specific context, return derived information; as such we describe any information that is not directly stored in the database. For example, a query that calculates the depth-of-inheritance metric returns derived information. Another example are queries that, given an XML node, return a human readable description of the program element represented by the node.

2.2 Integration Layer

The integration layer builds a graph from the results of the executed queries. A node of the graph is the representation of a software element or a derived information, returned by executing queries. Each node, which does not represent derived information, has a reference to the element it represents; this reference is used to ensure that each element of the database is represented by at most one node in the graph, even if the same element is selected by multiple queries. An edge represents a relation between two nodes exposed by executing a query; hence, the semantics of an edge is solely determined by the query. An edge always points from the node that was set as the context for the query evaluation (the source node) to the nodes that were returned by the evaluation (the target nodes).

Information about the visual appearance of an edge is specified in a query’s meta-data. For instance, it is possible to set the color and the description of an edge. In order to associate a meaningful description and visual representation with each node, the type of a node is determined by executing a special query with the node set as the context; the query maps elements in the XML database to identifiers, e.g., an element that represents a method declaration is mapped to the identifier MethodDeclaration. The type of a node determines the visual representation, i.e., the color and icon to use for visualization. Further, a query associated with each node type is executed to get a human readable description of the node’s XML representation.

All queries are defined with respect to a set of supported node types. A node can be used as the context of a query, if and only if the query explicitly supports the type of the node. While a node has exactly one type, a query can be defined with respect to multiple node types, e.g., a query to get the declaring class is well-defined for method declarations and field declarations.

2.3 Visualization Layer

The visualization layer visualizes the graph constructed by the integration layer and enables navigation through the software project. It is based on TwoMore, a tool originally designed for the manipulation and visualization of topic maps. Software is represented by a multigraph with ele-
ments as nodes and relations as directed edges between the nodes. Given a net of nodes shown on the screen, selecting a node will cause a pop-up menu to show up with a list of queries, available for this node, from which the user can choose to further navigate. The result of evaluating a selected query will be integrated into the existing net.

The visualization layer can be adjusted in various ways. For instance, one can choose between different layout algorithms. E.g., a hierarchical layout orders elements in a tree-like structure while a spring force layout places related elements in a concentric circle around the source node. It is also possible to provide new layout strategies. To keep parts of the graph structure fixed, one can explicitly assign a fixed place to one or more nodes. Fixed nodes have the advantage that they will not be rearranged when additional nodes are added to the net and therefore ease the comprehension. Furthermore, visual attributes such as color or form of nodes and edges can be customized.

The visualization layer of SEXTANT is tightly integrated with Eclipse to enable the navigation from a node to the source artifact.

3 Case studies

In this section, we present two case studies where we have used SEXTANT. In the first case study, we explore a small EJB project to demonstrate the need for the integrated comprehension strategy (R1). Furthermore, this case study also illustrates the need for cross-artifact support (R2) and extensibility (R4). In the second case study, we apply SEXTANT to understand parts of a complex application, namely the Steamloom Java virtual machine [3]. We analyze the dynamic weaving control flow and derive refactorings from the visual representation. This second case study will demonstrate how the explicit representation of all elements and relations (R3) as well as links between the graphical representation and the corresponding source elements (R5) are crucial for our needs. The size of the software explored in the second case study also provides first indications about SEXTANT’s performance in terms of memory and computation time.

3.1 Exploring an EJB project

As an introductory example, we present the usage of SEXTANT to explore a small EJB project. Consider a scenario in which a developer receives a bug report for an EJB component with the ejb-name CartBean, indicating problems with the transaction handling: Using this component resulted in a TransactionNotSupportedException. Given this information, the developer hypothesizes that the problem is causally related to the CartBean and starts searching for the class that implements the bean with the ejb-name CartBean using a corresponding query. The result is the class de.tud.CartBean, represented by the left-most node in Figure 2.

![Figure 2. Exploring for error tracking](image)

Next, the developer executes a query on this node to get all methods defined by the class that have the transaction attribute NotSupported. This query returns the node representing the getValue method. A further query to get all methods called by getValue does not return any further results.

After going back to the node of the CartBean class and executing another query to get all methods with the transaction attribute Required, the getText method is returned. Again, the developer wants to further explore the call graph and executes the called methods query on this node, which returns the toString method. After executing the same query once again for the toString method, the developer discovers that the toString method gets called from within toString, hence, finding a circle in the call graph. At this point, the developer immediately spots a severe problem: A method that always runs in a transaction context calls a method that does not support transactions.

Discussion

R1: This example illustrates the usefulness of a seamless integration of top-down and bottom-up comprehension. The basic top-down and bottom-up comprehension approaches are supported by means of the searching, respectively browsing facilities of SEXTANT.

Combining the two navigational approaches and switching between them is facilitated in that a single graph-based visualization is used to represent the results of a search as well as the elements a developer browses to. In our example, we started with a search for elements of the software with certain properties – implementation classes of beans with a certain ejb-name; subsequently, we continued the exploration by browsing the result of the search.

Furthermore, different views resulting from different queries and representing the system at different levels of abstraction, are visualized in a single graphical representation. For instance, the search query revealed the implementation of an EJB component while another query built the call graph for a method. Because those different views
provide complementary information, developers need to be able to navigate among them to completely understand an architecture [11]. With SEXTANT, it is possible to fuse different views by using different kinds of relationships in each step of the navigation.

R2: The EJB nature of our sample project illustrates the need for navigating across different kinds of artifacts. In our example, the queries used to search the bean implementation by its ejb-name and to browse to methods with specific transaction demarcations use information from Java class files as well as from the deployment descriptor, which is an XML file. Without an explicit support in software exploration tools, those kinds of relations have to be established 'manually'. First, this can be very time-consuming. Second, it runs contrary to the comprehension process, because not all elements and relations are visualized and one can easily get lost during the exploration [10].

R4: Last but not least, this first case study also demonstrates the need for extensibility. Common software elements and relations should be directly integrated in the tool. In our case, SEXTANT uses Magellan which comes with a large set of predefined, parameterized queries that implement standard search facilities for Java code, such as retrieving super-types/sub-types, retrieving all called methods or accessed fields, etc. On top of its base support, SEXTANT can be extended in three ways.

First, new kinds of artifacts can be added; support for new kinds of artifacts is provided by the underlying Magellan framework and described in our previous work [8].

Second, new element types can be added. Adding new node types involves a discussion as how to get from an XML element back to its corresponding source; since this feature heavily depends on Eclipse’s API a presentation is beyond the scope of this paper.

Third, SEXTANT can be enhanced by user-defined relations that enable developers to answer project, tool, or developer specific questions. After specifying a user-defined query, one can also specify meta-information, such as the name of the query and of the corresponding relation, as well as the supported node types. Such user-defined queries can then be used for navigating along new, user-defined kinds of relationships or to search elements with specific properties.

For illustration, consider how we made use of this feature in our first case study: For exploring our EJB application, we extended SEXTANT by writing the query shown below to determine the implementation class for a bean (with its name stored in the variable $ejb-name). In natural language this query reads as follows: First, save the name of the implementation class in the variable $class-name (lines 1-3). Then return the class with this name (line 4).

```xml
1. let $class-name := //ejb-jar/enterprise-beans/
2. (session|entity)=./ejb-name = $ejb-name]/
3. ejb-class/text()
4. return //bat:class[@name = $class-name]
```

3.2 Analyzing a Java VM

Steamloom [3] is a Java virtual machine with native support for aspect-oriented programming implemented as an extension of the Jikes Research Virtual Machine (Jikes RVM for short). Steamloom consists of roughly 200,000 lines of Java code, about 150,000 of which belong to the underlying Jikes RVM and other 30,000 belong to the bytecode toolkit BAT.

One of the features that Steamloom adds to the Jikes RVM is dynamic aspect weaving capabilities. The control flow inside the weaving component is complex; to understand it, we have applied some SEXTANT queries to the control flow that is initiated by the VM_AspectUnitRegistry.weaveIn() method. This method is responsible for weaving aspect functionality into one particular point in the application code. It calls one of three entry points of the CodeGenerator class: (a) generateCode(...), (b) generateAfterExecutionCode(...), or (c) generateAfterCallCode(...). The entire subsequent code generation control flow takes place inside the CodeGenerator class, by invoking several private methods.

By evaluating a query to find the called methods several times, we were able to visualize the entire weaving control flow within CodeGenerator as displayed in Figure 3. By observing certain patterns in the visual representation graph of the control flow, we were able to spot places in the weaving logic under exploration that had ‘bad smells’. Some of these observations led to refactorings after which we evaluated the same queries again to rebuild the modified control flow graph for weaving. The result is shown in Figure 4: The number of method calls has obviously decreased, and the overall control flow is much more clear. In the following, we briefly describe the particular ‘bad smell’ we discovered and the conducted refactorings. The various places in which refactorings were applied are marked by indexes in the two figures. Indexes in Figure 4 mark the results of refactorings applied to the respective locations in Figure 3.

One of the visual patterns can be seen at index 1 in Figure 3. The node representing the method generateNormalCode(...) has only one incoming and outgoing arrow, i.e., it is called from one method only and calls only one method, namely another version of generateNormalCode(...) with one more parameter. Switching to the source code, we recognize that this method merely forwards passing a default value to the called method. Since these methods are private, the control flow can be simplified in this case by deleting the forwarding version of generateNormalCode(...) and directly passing the default value of the second parameter from the actual call site.
Figure 3. Weaving control flow before refactorings

Figure 4. Refactored weaving control flow
The method generateJumpInstruction(...) (index 2) also has only one incoming arrow, meaning that it is only called from one site. Given that the method is rather trivial, it was inlined at that call site.

Another method, removeNOP(...), is called from each of the three entry point methods. Since it has no outgoing calls to other CodeGenerator methods, one might suspect that this method is used as a mere “service provider” that does not further contribute to the actual control flow. Indeed, by taking a closer look at the source code, we observed that it was invoked exactly once from each of the entry points. This invocation took place just before a list of instruction objects was returned to the caller, namely the weaveIn() method. Since removeNOP(...) also was rather trivial, this functionality was inlined in weaveIn(), right after the invocation of one of the three CodeGenerator entry points.

The initialize() method (index 4) is called by several generate*() methods. Given that initialize() does not accept any parameters and some of the calling methods invoke each other as well, this looked like a good opportunity for clarifying control flow. Indeed, initialize() is a simple set-up method that assigns initial values to some state variables of the CodeGenerator. It was easily possible to move the call to initialize() to the weaveIn() method before the actual code generation control flow is entered.

Each of the methods with indices 4, 5 and 6 is called from exactly one other method and could therefore be inlined at the respective call site.

Discussion

R3: This case study illustrates how an integrated view for searching and browsing avoids the need to switch between different tools and views, which often causes disorientation. Integrated views can help to create and retain the mental map and navigate through the system without getting lost. Furthermore, the case study demonstrates how the graph-based visualization of EXTANT directly represents the developer’s exploration path with all elements and relationships discovered in the exploration, whereby preserving referential integrity.

In fact, EXTANT extends the notion of “not getting lost” as introduced by Janzen and de Volder [10]. By using a graph for the visualization instead of a tree structure as in [10], complex relationships between different information sources become apparent. For example, browsing to the callees from the nodes representing the entry point methods during the exploration of the weaving control flow in Figure 3 results in the method initialize() in all three cases. Using a tree-based visualization, the method would appear in three different subtrees and the developer has to establish the relationship ‘manually’ by matching the names.

On the contrary, with the graph-based visualization of SEXTANT, the node for the initialize method appears only once in the exploration graph with incoming edges from all three entry points of the weaving process. Thus, graph-based visualizations can improve the comprehension of a software’s inner structure by making relationships among single elements explicitly visible. This explicit representation was crucial for understanding the control flow and ultimately led to refactorings and simplifications in the code.

R5: Finally, we discuss how the second case study motivates the requirement R5 from the introduction, concerning traceability between graphical representation and source code. The graphical representation was well-suited to detect visual patterns indicating possible structural “code smells”. However, to judge whether a refactoring is appropriate, one needs to look into the source code. For instance, we can see that the method generateNormalCode calls a homonymous method with one more parameter, but we pinpointed the method as a forwarder not until looking into the code. Due to SEXTANTS code link feature, one can synchronize the graphical representation with the code editor. Just after selecting any program element, the editor shows up the corresponding location in the source code. This enables to switch quickly between the different representations and further improves the comprehension, because one can also see low-level implementation details.

3.3 Limitations

During the work with SEXTANT we observed two limitations concerning the visualization layer. First, the graph-based representation does not scale. The larger the graph is, the less comprehensible it is. Second, there is not a single visualization appropriate for all maintenance tasks. In fact, each task poses specific requirements on the visualization.

To address these issues, we are currently working on a more flexible visualization layer than the simple graph-based visualization used in SEXTANT. We plan to provide means to let developers choose (a) the basic visualization technique, such as graphs or treemaps, (b) layout mechanisms, (c) 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.

4 Related work

Many tools have been developed to support the understanding of software systems. One category of software comprehension tools focus on the visualization facilities [6, 11, 15, 23, 24]. Two well-known examples are Rigi [15]
and SHriMP [23]. 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 multi-window approach. SHriMP, a tool based on Rigi, provides an alternative visualization. All subsystems are represented in a single view using nested graphs. Along those, developers can navigate down to the source code. Besides this, well-known visualization techniques such as fisheye-view or pan and zoom are also available.

In contrast to SEXTANT with its lightweight visualization, both tools provide more complex visualization techniques. But, recent studies revealed that developers are often swamped with too many elements [12, 24] and too complex visualizations [25]. For example, the existence of multiple, non-integrated views can cause disorientation as in case of Rigi, whereas the SHriMP visualization can result in an information overload. Our approach differs in such that not the whole system is visualized, but the developer explores software elements of interest step by step. Though, Rigi and SHriMP provide support for source code navigation using hyperlinks and for context navigation, the 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 seamlessly to the source code.

Other tools aim at combining the two navigational styles searching and browsing — a prerequisite to support the integrated model of software comprehension. Examples of those tools are Hy+ [14], Ciao [5], The Searchable Bookshelf [19], and SPOOL [17]. 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 to search or browse across relationships, one cannot fuse different views. If a developer uses for instance SPOOL and starts its 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 make the results of the former query as starting points for a new query, one looses the exploration path which is essential to build up a mental map of the software system [7].

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 [16] is a tool to create and manipulate representations of concerns. Developers can browse 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 element representation.

SEXTANTS capabilities to navigate and visualize program elements are more advanced compared toFeat with its fixed program model. We provide means to search and browse along 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 mean to make concern descriptions explicit using concern graphs. We will investigate the use of concern graphs to persist exploration graphs.

The tool most similar to SEXTANT is JQuery [10]. It combines the advantages of query-based tools and hierarchical browser tools. Queries provide means to search for elements in the system on the one hand, and to explore code in terms of different kinds of relationships on the other hand. The results are visualized in a tree-based, hierarchical representation. Each resulting element can be used as source for a new query. The results of this query form a subtree of the source element. Thus, the whole tree is an explicit representation of the exploration path. However, due to the hierarchical nature of JQuery’s visualization, a single element can occur several times in different subtrees of the exploration. Each occurrence symbolizes a relationship between the element itself and its parents. However, to see all relationships of the elements the developer has to derive this knowledge manually by searching for all its occurrences in the tree. Another shortcoming is that JQuery is only capable to explore the Java structures in a software system. The tool is not intended to integrate other kinds of artifacts, which restricts the applicability in modern software projects.

In contrast to JQuery, SEXTANT uses a graph-based representation which improves the comprehension of the relationships between the systems 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 not only Java source code is used.

A different approach of software exploration, namely back-packing exploration, is described by Favre [9]. 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 back-packing framework provides a generic successor interface with a single method returning all related elements for a given one. This enables the integration of various in-
formation 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 meta-model 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 back-packer adds different extractor tools to integrate a new kind of artifact on-the-fly. Thus, the back-packer approach is particularly suitable for exploring new "software landscapes" with different, non-common kinds of artifacts. However, SEXTANT is more appropriate in case of common artifact kinds because the query facilities enable to define relationships over various artifacts in one step, instead of having to explore those relationships incrementally.

5 Summary and Future Work

In this paper, we presented SEXTANT, a software exploration tool, the most important features of which are summarized in the following, indicating their relevance for the software comprehension process.

Support for the integrated comprehension model. SEXTANT’s advanced search capabilities can be used to verify hypotheses about the software by formulating queries to search for evidences for these hypotheses in the program code and/or other software artifacts. Queries are also used to browse between elements along different kinds of relationships. By starting with low-level program elements, e.g., field accesses, method calls, method declarations etc., and browsing to higher-level program elements such as classes, or packages, developers mentally create a more and more abstract view of the software in a stepwise manner. Thus, both top-down and bottom-up comprehension are supported. Furthermore, a rapid and seamless switching between different views, each presenting the system at different abstraction levels is supported. This feature is crucial for the integrated model of software comprehension [19]. The integrated, graph-based view keeps the single steps of an exploration process connected, which enables the efficient combination of searching and browsing without getting lost. We illustrated this feature in both examples.

Cross-artifact comprehension. SEXTANT supports the exploration and comprehension of software projects that involve several kinds of artifacts, such as source code, XML deployment descriptors, build scripts and configuration files. This is due to the Magellan framework which serves as the back-end of SEXTANT [8]. As we have seen in the exemplary EJB project, this feature is important, because most real projects are characterized by their use of a multitude of artifacts.

Explicit structure. The view technique used in SEXTANT visualizes the exploration path and all relations between discovered elements are depicted explicitly. In this way, developers obtain new knowledge about the structure of the software. This was illustrated in the Steamloom example.

Extensible architecture. SEXTANT provides a rich set of predefined queries which can be used for searching and/or browsing. However, to explore software of a specific domain or using a particular technology, one can adjust the tool by adding new kinds of artifacts, node types, and kinds of relationships as shown in the EJB example.

Navigation to source artifacts. In SEXTANT, each element is tied to its corresponding source fragment of the software. This is important, because once developers have discovered a property of their software using exploration, it is extremely helpful to map the discovery onto the source artifact, which, e.g., might need to be refactored as the result of the discovery. This feature was illustrated in the Steamloom example.

We conclude that SEXTANT supports all requirements for exploration tools defined in the introduction. Individual requirements are also supported by other comprehension tools. But, as discussed in the related work session, to the best of our knowledge, none of the related tools supports the combination of features that are provided by SEXTANT.

For evaluating its effectiveness and performance, we used SEXTANT for exploring an EJB project and a virtual machine implementation. The results of this evaluation were very encouraging not only in terms of software comprehension support provided but also in terms of performance. It is notable that all queries were, despite the large underlying database — about 200,000 LOC for the entire Steamloom project – evaluated in about one second at most. We have not conducted detailed performance measurements, but the speed experienced during working on Steamloom was in all cases satisfactory.

There are several important areas of future work. First, we will further evaluate SEXTANT in the context of real reverse engineering projects. Besides evaluation purposes, the goal will also be to collect more experience with common tasks in reverse engineering projects and their needs on search and navigation capabilities of the tool. This knowledge will serve to extend the set of predefined search and navigation capabilities of SEXTANT.

The second area of future work aims at further optimizing the support provided for constructing and preserving a developer’s mental map of the software during exploration. We intend to keep the light-weight visualization technique of SEXTANT because it proved helpful in building mental maps of software, while avoiding potential disorientation caused by complex visualization techniques [25, 24]. How-
ever, customizable use of colors, different forms, or different layout mechanisms could further improve the cognitive processes. As already mentioned, SEXTANT is capable of exploiting different strategies, e.g., for the layout of the elements, or their representation. However, further investigation of the visualization strategies with respect to their effect on the comprehension process is needed.

Another important issue that needs further investigation concerns scalability. During the exploration process, developers often take meanders, go back to a previous element and continue exploration from this point. Those meanders can bloat the element graph while not improving the program understanding. We will investigate possibilities to filter out elements that do not contribute to the understanding process at a certain point in time in order to facilitate focusing the attention to the important elements.

Acknowledgement

This work has been funded in part by a Darmstadt University of Technology fellowship. The authors would like to thank Siniša Đukanović and Mario Vekić for implementing parts of the tool. We are also grateful to the four anonymous reviewers for their helpful comments.

References