Supporting developers with natural language queries

Würsch, M ; Ghezzi, G ; Reif, G ; Gall, H C

Abstract: The feature list of modern IDEs is growing steadily and mastering these tools becomes more and more demanding, especially for novice programmers. Despite their remarkable capabilities, IDEs often still cannot directly answer the questions that arise during program comprehension tasks. Instead developers have to map their questions to multiple concrete queries that can be answered only by combining several tools and examining the output of each of them manually to distill an appropriate answer. Existing approaches have in common that they are either limited to a set of predefined, hardcoded questions, or that they require to learn a specific query language only suitable for that limited purpose. We present a framework to query for information about a software system using guided-input natural language resembling plain English. For that, we model data extracted by classical software analysis tools with an OWL ontology and use knowledge processing technologies from the Semantic Web to query it. We also present a case study that demonstrates how our framework can be used to answer queries about static source code information for program comprehension purposes.

DOI: https://doi.org/10.1145/1806799.1806827

Supporting developers with natural language queries

Würsch, M; Ghezzi, G; Reif, G; Gall, H C

Supporting developers with natural language queries

Abstract

The feature list of modern IDEs is growing steadily and mastering these tools becomes more and more demanding, especially for novice programmers. Despite their remarkable capabilities, IDEs often still cannot directly answer the questions that arise during program comprehension tasks. Instead developers have to map their questions to multiple concrete queries that can be answered only by combining several tools and examining the output of each of them manually to distill an appropriate answer. Existing approaches have in common that they are either limited to a set of predefined, hardcoded questions, or that they require to learn a specific query language only suitable for that limited purpose. We present a framework to query for information about a software system using guided-input natural language resembling plain English. For that, we model data extracted by classical software analysis tools with an OWL ontology and use knowledge processing technologies from the Semantic Web to query it. We also present a case study that demonstrates how our framework can be used to answer queries about static source code information for program comprehension purposes.

Supporting Developers with Natural Language Queries

Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall
{wuersch,ghezzi,reif,gall}@ifi.uzh.ch
s.e.a.l. – software architecture and evolution lab
Department of Informatics
University of Zurich, Switzerland

ABSTRACT

The feature list of modern IDEs is steadily growing and mastering these tools becomes more and more demanding, especially for novice programmers. Despite their remarkable capabilities, IDEs often still cannot directly answer the questions that arise during program comprehension tasks. Instead developers have to map their questions to multiple concrete queries that can be answered only by combining several tools and examining the output of each of them manually to distill an appropriate answer. Existing approaches have in common that they are either limited to a set of predefined, hardcoded questions, or that they require to learn a specific query language only suitable for that limited purpose. We present a framework to query for information about a software system using guided-input natural language resembling plain English. For that, we model data extracted by classical software analysis tools with an OWL ontology and use knowledge processing technologies from the Semantic Web to query it. We use a case study to demonstrate how our framework can be used to answer queries about static source code information for program comprehension purposes.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments, Interactive environments, Programmer workbench*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Query formulation*; I.2.3 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Semantic networks*

General Terms

Human Factors, Languages

Keywords

Software evolution, software maintenance, source code analysis, semantic web, natural language, conceptual queries, tool support

ICSE '10, May 2-8 2010, Cape Town, South Africa

1. INTRODUCTION

Program comprehension plays an important role when performing software engineering activities on large bodies of source code. Both industry and research therefore have aimed to integrate various tools into modern integrated development environments (IDEs) to support software engineers in understanding a software system during their daily development and maintenance tasks.

Especially for novice developers, mastering all the powerful features that are delivered by an IDE, such as Eclipse or Visual Studio, requires a great deal of learning effort. When solving a program comprehension task, developers usually have particular questions in mind, such as *"Where is this method called?"* or *"What fields are declared as being of this type?"* [33, 34]. Unfortunately, such questions often can not be answered directly using existing functionality offered by IDEs. Instead, as explained by De Alwis and Murphy in [9], developers first need to map these *conceptual queries* to one or several *concrete queries*. Even if a particular conceptual query is directly supported, novice programmers are often not yet aware of the capabilities of their IDE. For example, although experienced developers can easily answer *"Where is this method called"* with Eclipse, newcomers still need to be aware that the *"Find references..."* feature, hidden in a context menu, is what they are looking for.

Existing approaches to enable the integration of different information sources often do not allow developers to formulate ad-hoc queries. Instead, they need to be explicitly configured to enable new queries. On the other hand, query languages, such as CodeQuest [17] or JQuery [23], allow developers to formulate queries about software engineering artifacts. These languages are usually based on a SQL- or Prolog-like syntax and effectively using them requires learning effort. According to Chowdhury, however, *"the most comfortable way for a user to express an information need is as a natural language statement."* [6]. Henninger even suggests that constructing effective natural language queries is as important or more important than the retrieval algorithm used [20].

In this paper, we present a framework that allows software engineers to use *guided-input natural language* strongly resembling plain English to query for information about a software system. For that, we combine software evolution data provided by EVOLIZER, our platform for software evolution analysis, with Semantic Web technologies for knowledge processing. We focus on queries concerning static source code information, such as *"How often is this field accessed?"* or *"What are the subclasses of this class?"*, to demonstrate the potential of our approach; but including more data from various software repositories and tools is straight-forward, as we will explain in detail in this paper.

The remainder of this paper is structured as follows: In Section 2 we give an introduction to the Semantic Web and discuss

the knowledge processing technologies that we use throughout the paper. We present our framework to query static source code information with (quasi) natural language in Section 3. Section 4 provides a case study evaluation of our approach. Section 5 discusses existing work related to our approach and Section 6 concludes the paper.

2. BACKGROUND

The technologies originally developed for the Semantic Web have been proven useful whenever knowledge has to be processed by machines. In this paper, we exploit them to bridge the gap between more classical software analysis tools and natural language query interfaces.

Tim Berners-Lee *et al.* [2] define the Semantic Web as *"an extension of the Web, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."* Following this definition, this semantic extension enriches the Web with meta-data describing the semantics of the webpages in a computer-processable way. Before the webpages can be described with meta-data accordingly, an *ontology* has to be defined for the domain of discourse. An ontology formally describes the concepts (classes) found in the domain, the relationships between these concepts and the properties used to describe them [15]. For example, in the domain of an online record shop, we define concepts, such as *Composer*, *Album*, and *Track*; the relationships *composed by* and *has track*, and the properties *has play-time* and *has title*. The meta-data description of a CD is then able to give the data a well-defined meaning, using the concepts, relationships, and properties defined in the ontology. In the software engineering domain, we define concepts, such as *User*, *Developer*, *Bug*, *Module*; relationships, such as *reports bug*, *fixes bug*, and *is assigned to bug*. Since the Semantic Web describes this information based on formal semantics, data can be exchanged among two applications that support the same ontology, even if they were not meant to interoperate in the first place.

To bring the vision of the Semantic Web into being, the research community came up with a number of standards, W3C recommendations, development frameworks, APIs, and databases. The Resource Description Framework (RDF) [26] is the data-model for representing meta-data in the Semantic Web. The RDF data-model formalizes meta-data based on *subject – predicate – object* triples, so called RDF statements. RDF triples are used to make a statement about a resource of the real world. A resource can be almost anything: a bug report, a person, a Web page, a CD, a track on a CD, etc. Every resource in RDF is identified by a Uniform Resource Identifier (URI) [1].

In an RDF statement the subject is the thing (the resource) we want to make a statement about. The predicate defines the kind of information we want to express about the subject. The object defines the value of the predicate. In the RDF data-model information is represented as a graph with the statements as nodes (subject, object) connected by labeled, directed arcs (predicate). The query language SPARQL [32] can be used to query such RDF graphs.

RDF is domain-independent in that no assumptions about a particular domain of discourse are made. It is up to the users to define specific ontologies in an ontology definition language, such as the Web Ontology Language (OWL) [10].

OWL enables the use of description logic (DL) expressions to further describe the relationships between classes and to restrict the use of properties [28]. For example, two classes can be declared to be disjoint, new classes can be built as the union/intersection of others, or the cardinality of a property can be restricted to define how often a property can be applied to an instance of a class.

In addition to the W3C recommendations, the Semantic Web community developed tools to process RDF meta-data. *Jena*[1] emerged from the *HP Labs Semantic Web Program* and is a Java framework for building applications for the Semantic Web. It provides a programmatic environment for RDF and OWL.

In this paper, we do not contribute directly to the Semantic Web, but we exploit the technologies introduced above to describe and process data. In short, we model software evolution data with an OWL ontology. Then we take, for example, the static source code information that was extracted with our EVOLIZER toolset, and represent it as an RDF graph that is based on this ontology. RDF graphs, in contrast to relations in a relational model, consist of {*subject, predicate, object*}-triples which are close to the natural English sentence structure. This enables the transformation from natural English queries into SPARQL queries which can be issued on the RDF graph. In the remainder of the paper, we describe in detail how the RDF graph is generated and how this knowledge representation is exploited to support software developers.

3. APPROACH

Our vision is to provide a convenient and intuitive interface that allows software engineers – and especially newcomers, who are not yet virtuous with their IDE or command line tools, such as grep and awk – to leverage information sources about various aspects of a software system under development. In particular we want to overcome some of the limitations that existing approaches suffer from: We do not want to restrict developers to a set of predefined queries and we do not want them to learn a specific query language for that limited purpose. Instead, we want developers to be able to use a query language that they are already very familiar with: natural language. A natural language interface to an IDE relieves especially novice programmers from the cognitive burden of translating a conceptual query to a concrete one, which might involve navigating through nested or multi-level menus.

In the following, we briefly introduce EVOLIZER, our platform for software evolution analysis. For the sake of this paper, we focus on the infrastructure that is needed to give developers the possibility to query static source code information from within Eclipse. However, we want to emphasize again that our approach can be generalized to many other domains in the software evolution context.

3.1 Evolizer

EVOLIZER[2] basically stems from the idea of having a Release History Database (RHDB) [12] that integrates information originating from various repositories, such as CVS and Bugzilla, in a single database. It facilitates many common preprocessing steps [38] that are necessary when mining such software archives and, in that context, it is comparable with *Kenyon* of Bevan *et al.* [5] or *eROSE* of Zimmermann *et al.* [39].

Over the years EVOLIZER has advanced from a set of data importers and preprocessors to a platform for various tools that aid developers during their daily maintenance and development tasks. Realized as Eclipse plug-ins, the functionality of EVOLIZER is available at developers' fingertips in a state-of-the-art IDE and incorporates applications that emerged from ideas of the software evolution research community, as well as more classical approaches to support, for example, program understanding. Some of the tools that are built upon EVOLIZER, such as CHANGEDISTILLER [13, 14], follow the original idea of leveraging historical data. Others do not

[1] http://jena.sourceforge.net/

[2] http://www.evolizer.org/

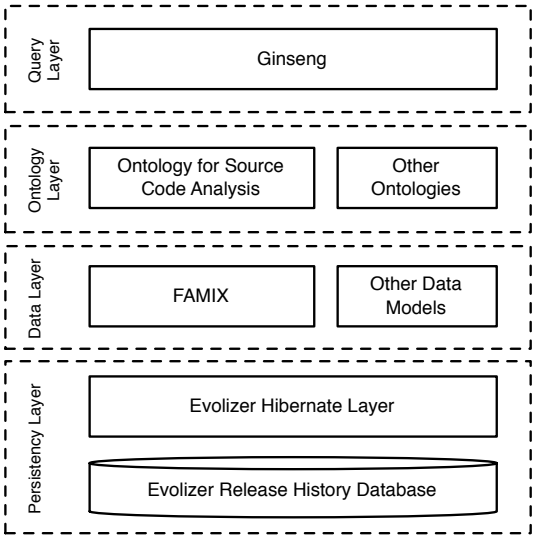

Figure 1: The four layers of Evolizer.

make use of any evolutionary data at all. Instead, for example in case of DA4JAVA [31], they analyze source code that is currently under development within the IDE.

Figure 1 gives an overview on the four layers of EVOLIZER that are relevant for this paper. The persistency layer gives access to facts about a system in a convenient way and provides application support for other EVOLIZER plug-ins to persist settings, query results and so on. In the following, we provide detailed insight into the other three layers: In Section 3.2 we describe the *data layer* consisting of a set of data models and data importers. Section 3.3 describes the *ontology layer* that enhances the data layer with formally described data semantics. In Section 3.4 we show how existing Semantic Web query technology can be used to query an ontology-based knowledge base with quasi-natural language. Section 3.5 sums up how the three layers play together to allow a developer to access facts about her source code in a convenient and intuitive way.

3.2 Evolizer Data Layer

EVOLIZER provides a set of data models to represent software evolution data along with adequate importer tools to obtain this data from software project repositories. Extending existing data models and data importers, or adding new ones, is straight-forward. For the approach that we present in this paper, we use a tool that was implemented on top of our platform to perform static source code analysis and store the result in our EVOLIZER RHDB.

To add new data, we first identify the key concepts that we want to analyze (in case of analyzing source code: packages, classes, methods, accesses, invocations, etc.) and create a data model accordingly. We use the plug-in extension facilities of Eclipse to make EVOLIZER aware of its data models, so that they can later be accessed by other EVOLIZER plug-ins. For the approach presented in this paper, we plug in a custom-tailored implementation of the FAMIX model [35] to represent facts about source code. Eventually, we need a data importer to extract and store information into the data model that we have registered. In case of our example, this is a parser that extracts the relevant facts from source code under analysis. The FAMIX source code meta-model and the fact extractor that we use are covered in more detail in the next section.

Data models in EVOLIZER are implemented using Java classes

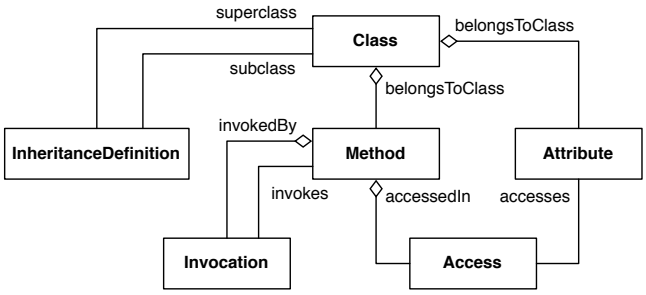

Figure 2: Core of the FAMIX model by Tichelaar *et al.*

annotated with object-relational mapping meta-data according to the Enterprise JavaBeans 3.0 (EJB3.0) specification.[3] Persistency is provided through Hibernate,[4] a well-known high performance object/relational persistence and query service. As denoted earlier, EVOLIZER maintains a list of all registered data models and provides means to other EVOLIZER plug-ins to access the evolutionary information of a system via a convenient API.

3.2.1 Evolizer FAMIX

The FAMIX meta-model provides a language-independent representation of object-oriented source code [35] and we use it in EVOLIZER whenever an analysis requires static source code information. An overview of the core model is given in Figure 2. It specifies the entities that can be extracted directly from source code.

The model defines important object-oriented relations as classes. For example, *Invocation* is explicitly modeled as a class instead of using a self-aggregation for *Method* (which would be implemented in Java as a collection of *Method*s as an Attribute of the class *Method*). This yields several benefits for us; the most important one is that we can map *Invocations* directly to the EVOLIZER RHDB and query them explicitly later. For example, we can retrieve all *Invocations* that fulfill certain properties, such as that they point to a particular method we are interested in, using HQL – the Hibernate Query Language:

```
from Invocation as invocation
    join invocation.callee as callee
where
    callee.name='addChart';
```

The results of such a query can then be used to, *e.g.,* provide a dependency analysis or visualization. EVOLIZER therefore already provides basic SQL-like query access, with features comparable to existing query languages for software analysis, to those that are familiar with both, HQL and the data available in the RHDB. On the other hand, the deficiencies of existing query languages also apply here; for developers that have no deeper knowledge of HQL and the underlying data structures, the information is hardly accessible through the data layer. In Section 3.3, we therefore describe how we add another layer to EVOLIZER to enable natural language interfaces.

We rely on a custom implementation of the FAMIX model, realized according to the procedure that we have outlined in Section 3.2. To populate an instance of the model with concrete data from source code under analysis, we use ZBINDER by Pinzger *et al.* [30]. ZBINDER builds upon the Java Development Tools (JDT)[5]

[3] http://java.sun.com/products/ejb/
[4] http://www.hibernate.org/
[5] http://www.eclipse.org/jdt/

of Eclipse. The JDT parser alone fails in resolving cross references, such as method calls and attribute accesses of statements that contain a compile error – which is often the case when the code is incomplete or referenced libraries are missing. ZBINDER overcomes this limitation in most instances by gathering additional information stored in the abstract syntax tree and using sophisticated heuristics to reconstruct unresolved method calls.

Static source code information for a software system can either be extracted from past releases that are stored within the EVOLIZER RHDB, or directly from a project that is currently under development in the workspace of Eclipse. ZBINDER can even be registered as a *builder* for a project, so that it gets notified every time a change is made to the source code.

The data layer of EVOLIZER provides a strong foundation for most of the classical software evolution and engineering analyses, especially when they depend on database performance, *e.g.,* interactive software visualizations. Knowledge processing tasks and tool integration, on the other hand, would benefit from formally defined data semantics that the data layer alone can not provide.

In the next section, we demonstrate by example how we overcome this limitation by adding an ontology layer to our platform.

3.3 Evolizer Ontology Layer

The EVOLIZER Java implementation of the FAMIX meta-model does not explicitly describe the formal semantics that is needed for automatic knowledge processing tasks such as query answering. For example, we can not define that there is an inverse relation between the property *declares Method* of a *Class* and the property *is declared in Class* of *Method*. If we are able to explicitly state the formal semantics then, every time we make a statement, such as *A declares B*, a semantic reasoner would be able to automatically infer that also *B is declared in A*. The Web Ontology Language OWL allows us to use description logic expressions to describe such relationships and to reason about them.

To take advantage of Semantic Web technologies, we added an additional layer on top of the EVOLIZER data layer by defining an OWL ontology that represents the FAMIX meta-model in terms of OWL classes, relationships and properties. This *source code ontology* is a subset of our software evolution ontology called `SEON`.[6] Instance data is represented by RDF graphs. This way we get a knowledge base whose formal semantics enables automated processing. An overview of the ontology is shown in Table 1. The full ontology covers many more concepts, such as *interfaces*, *local variables*, *type casts* and usages of the *instanceof*-operator, as well as *exceptions*, but for the sake of this paper, we only focus on key concepts.

Class: Class	**Class: Method**
→ hasMethod Method	→ accessesAttribute Attribute
→ hasAttribute Attribute	→ hasParameter Parameter
→ isReturnTypeOf Method	→ invokesMethod Method
→ isSubclassOf Class	→ hasReturnType Class
→ isSuperclassOf Class	→ isInvokedByMethod Method
→ hasName String	→ isMethodOf Class
	→ hasName String
Class: Attribute	
→ isAttributeOf Class	**Class: Parameter**
→ isAccessedByMethod Method	→ isParameterOf Method
→ hasName String	→ hasName String

Table 1: Simplified Version of the Evolizer Ontology for Source Code Analysis.

To populate the knowledge base, we need to map our Java implementation of the FAMIX meta-model to the OWL ontology. This mapping is done via a custom Java annotation `@rdf`. We add an annotation with the URI of the according OWL class to the signature of each Java class that has a counterpart in the ontology. Similarly, we annotate each Java method that should be mapped to a corresponding OWL relation or property name. We use Java reflection to automatically generate RDF statements from Java instances. This approach is similar to – and partially inspired by – the *so(m)mer*-project,[7] an Object-to-RDF mapping layer that uses annotations in the spirit of Hibernate. In Listing 1 we show an example of an annotated Java class. We have omitted the EJB3.0 annotations for persistency that are used by the data layer of EVOLIZER.

```
@rdf("http://evolizer.org/ont/java#Class")
public class FAMIXClass {

  @rdf("http://evolizer.org/ont/java#hasName")
  public String getName() {
    // ...
  }

  @rdf("http://evolizer.org/ont/java#hasMethod")
  public Set<FAMIXMethod> getMethods() {
    // ...
  }

  /* attributes, setter methods, and
     additional behaviour */
}
```

Listing 1: Java class annotated with @rdf.

For the following discussion, we introduce the namespace prefix `evo:` as shortcut for `http://evolizer.org/ont/java#`. In the example, the Java class `FAMIXClass` is annotated with the URI `evo:Class` and therefore, for every instance of the `FAMIX-Class`, an instance of the OWL class `evo:Class` is generated in the RDF graph. This is done by creating a resource in the graph and adding a `rdf:type` property with `evo:Class` as value. The URI that represents the resource is generated using the unique database id maintained by the data layer of EVOLIZER.

In addition, for each annotated method in the Java class, a property is added to the resource. The return value of the annotated method is used as value of the property in the graph. Since the `getName()`-method has the return type `String`, the value is added as a literal. For the `getMethods()`-method the return type is a set of instances of `FAMIXMethod`. Since the return type is a `Set`, we add a property for each element in the set. The elements in the set are instances of the `FAMIXMethod` class (which is also annotated with `@rdf` annotations). Therefore, we trigger the generation of the RDF statements for the `FAMIXMethod` class as well, repeating the process described above.

An excerpt of the generated RDF graph is shown in Figure 3. Ellipses represent resources in the graph. The labels in the ellipses are the URIs that uniquely identify the resources. The labeled arcs represent the properties that relate the subject and the object of a RDF statement. Finally, literals are depicted as rectangles. In addition to the example that we have outlined above, we list also additional subgraphs that belong to the *version control* and *issue tracking* ontologies in Figure 3. This gives an impression on how we integrate different kinds of data sources, although we focus solely on knowledge covered by our code ontology in this paper.

In our Java-to-OWL mapping, we have to take a special case into account. Not all Java classes have counterparts among OWL

[6] SEON is available for download at http://www.evolizer.org/

[7] https://sommer.dev.java.net/

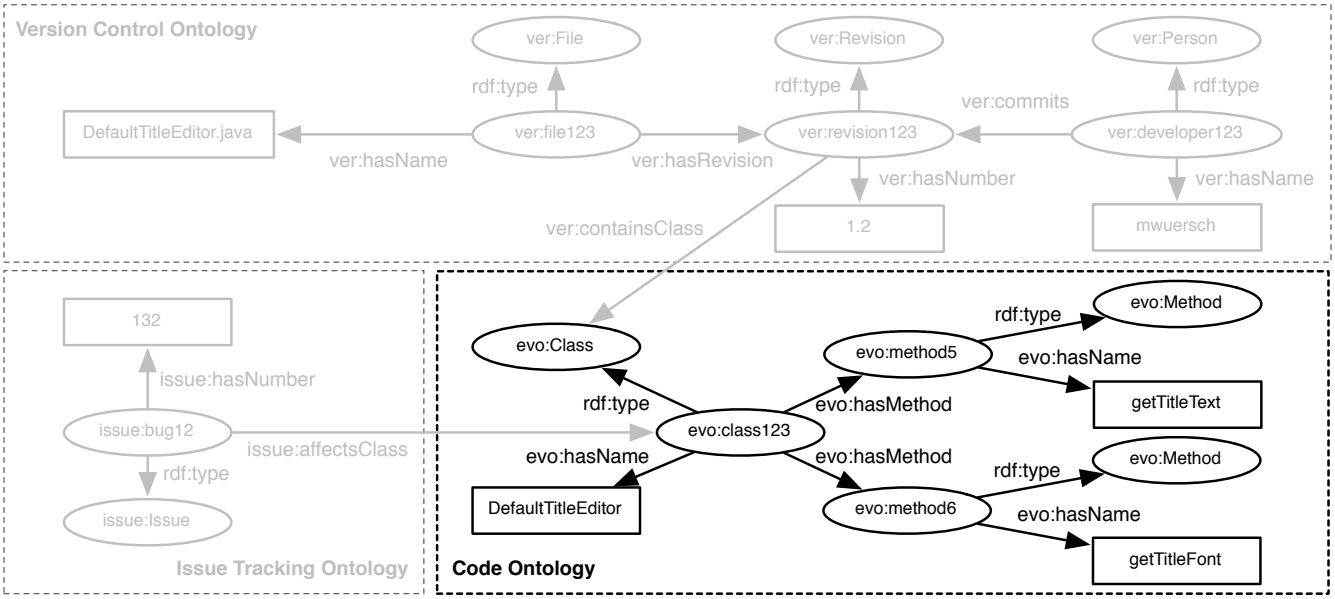

Figure 3: Excerpt of a generated RDF graph.

classes. Java classes that explicitly model relationships are usually modeled as properties or relations in the ontology. In the RDF graph, they are represented as predicates. For example, the *Invocation*-class in the FAMIX model is modeled as *invokesMethod*-property in our ontology. We overcome this clash of paradigms by making it possible to mark a Java class explicitly to model a property by setting the flag `isPredicate` to `true`. In addition, we introduce two additional Java annotations `@subject` and `@object` to specify that, for example, in case of `Invocation`, the method `getCaller()` returns the subject and `getCallee()` the object of the RDF statement. The value of the `@rdf`-annotation is then considered to be the URI of an OWL-property, rather than of an OWL-class. This renders our mapping approach much more flexible, especially when the underlying Java class models were influenced by a relational view. An example of a class that models a property is given in Listing 2.

```
@rdf(
   isPredicate=true,
   value="http://evolizer.org/ont/invokesMethod"
)
public class Invocation {

   @subject
   public FAMIXMethod getCaller() {
      // ...
   }

   @object
   public FAMIXMethod getCallee() {
      // ...
   }

   /* attributes, setter methods, and
      additional behaviour */
}
```

Listing 2: Java class that models a property.

3.4 Evolizer Query Layer: Natural Language Querying with Ginseng

So far, we have discussed how we import static source code information (and facts about the evolution of a system in general) into our EVOLIZER RHDB. We also showed how we use ontology information to augment this data. We explained that we rely on the established industry standard OWL/RDF which enables us to leverage the potential of EVOLIZER with a plentitude of existing tools and frameworks for knowledge processing from academia, as well as from industry.

One such tool is Ginseng, a **g**uided **i**nput **n**atural language **s**earch **eng**ine, that was presented by Bernstein *et al.* in [3]. Ginseng benefits from the fact that ontologies are described in terms of triples of *subject*, *predicate*, and *object*. This structure strongly resembles the way how humans talk about things. It can be exploited to use quasi-natural language queries for accessing any OWL/RDF knowledge base. Ginseng is lightweight compared to classical full natural language interfaces since it uses no predefined vocabulary and queries are not interpreted logically or syntactically. Instead, the vocabulary is derived from the OWL knowledge base itself, *i.e.,* from labels of the instance data and from the identifiers that were used to define the ontologies. Optionally it is possible to add synonyms by annotating the ontology with Ginseng tags.

Ginseng uses a multi-level grammar consisting of a static part that defines basic sentence structures and phrases for English questions, and a dynamic part that is generated when an ontology is loaded. The static part additionally contains information on how to translate query sentences from quasi natural language to SPARQL. To generate the dynamic part of the grammar, the ontology is loaded into a Jena inferencing model and for each OWL class, individual (instance), object property, and data type property, a grammar rule is generated.

The full grammar is then used by Ginseng to guide its users by offering an auto-completion feature, *i.e.,* it presents a popup box with suggestions on how to complete the word that the developer is currently typing into the free-form input field. This limits the questions a developer can ask to a certain extent but prevents her from entering queries that are grammatically incorrect. Once the com-

plete query is entered and concluded by a question mark, it is translated by Ginseng into SPARQL statements and executed against the knowledge base maintained by Jena. The results of the query are then presented to the developer.

Consider again the example graph given in Figure 3. If we want to query for all the Methods that are declared in the class `DefaultTitleEditor`, we could ask:

What are the methods of DefaultTitleEditor?

That question does not need to be reformulated to a more formal query – it is accepted by Ginseng as it is listed above and developers additionally receive guidance in query composition when they start to type. By *guidance*, we mean that, in case of our example, as soon as the letter *'W'* is typed, all the words starting with that letter are listed in a drop-down menu (see Section 4.1 for a full, illustrated example). Ginseng continues to do so until a complete and valid sentence (in terms of that it satisfies the grammar rules) was entered and will then automatically continue with translating the question into the following SPARQL query:

```
SELECT ?methods
WHERE {
   ?methods <rdf:type>      <evo:Method> .
   ?class   <evo:hasMethod> ?methods .
   ?class   <rdf:type>      <evo:Class> .
   ?class   <evo:hasName>   "DefaultTitleEditor" .
}
```

When the query above is executed, the graph pattern consisting of the four triple patterns in the `WHERE`-clause is matched against the triples in the RDF graph, and returns the bindings for the variables in the `SELECT`-clause. In SPARQL, variables are indicated by the prefix "?". The predicate and the object of the first triple are fixed values, so the pattern is going to match only triples with those values. Within a graph pattern a variable must have the same value no matter where it is used, so the query above only returns a binding for `?method` if the variables called `?method` in the first and second triple have the same value; as well as the variables called `?class` in the second, third, and fourth triple.

For more details on Ginseng and its underlying technology, we refer to [3].

3.5 Wrapping up: The Integration of the three Layers of Evolizer

When we want to query for facts about source code, we tell the data layer of EVOLIZER to parse the currently selected project in the Eclipse workspace (alternatively it is possible to process the code of a past release stored within the RHDB). The most convenient way to trigger this process is to add a EVOLIZER *Nature* to an Eclipse project. Along with the *nature*, a builder is then assigned to the project that automatically re-builds the FAMIX-model every time a change to the source code is made. Re-building the FAMIX model means that the source code is parsed by `ZBinder` which creates instances of `FamixClass`, `FamixMethod`, `Invocations`, and so on, according to the facts that it finds within the source code. Then it stores these instances into the RHDB.

The data is then passed to the ontology layer which translates it according to the `@rdf`-annotations and the OWL description of the *Evolizer Ontology for Source Code Analysis* into a Jena Ontology model.

This model is then analyzed by Ginseng and, subsequently, available to the developer for querying in natural language. The results are presented to the developer in an Eclipse view, similar to that provided by Eclipse itself for displaying Java search results. Since we also keep track of source code locations in our model, the developer can easily navigate from the results view directly to the corresponding source code.

Next, we provide a case study to demonstrate how our prototype implementation of the framework described above can be used by a developer to answer common questions during daily program comprehension tasks.

4. CASE STUDY

In our case study, we demonstrate by the example of the open-source library JFreeChart[8] that our framework can be used to answer the most common program comprehension questions that arise during software evolution tasks [9]. We do not focus on evaluating the quality of the query results – as we have explained throughout the preceding sections, the data importers are not the key contribution of this paper and can be exchanged easily thanks to EVOLIZER's plug-in architecture. Instead, we show that, compared to existing tools, developers are given more flexibility when composing queries with our approach: they can formulate queries conveniently using different variations of natural language sentences.

In [24], Kaufmann *et al.* presented a usability study with 48 users, evenly distributed over a wide range of backgrounds and professions, including software development. The study incorporated four query interfaces (including Ginseng) featuring four different query languages that demonstrated the usefulness of natural language interfaces for casual end-users. Their experiment was based on geographical data encoded in an OWL knowledge base. Kaufmann *et al.* found that:

> *"(1) With full-sentence questions, users can communicate their information need in a familiar and natural way without having to think of appropriate keywords in order to find what they are looking for. (2) People can express more semantics when they use full sentences and not just keywords." [24]*

Although we did not yet conduct an extensive user study in the software engineering domain, we claim that the results of this study are, to some extent, applicable to our setting. It is reasonable to assume that the domain of the knowledge that we query can be neglected, compared to the influence of the professional background of the users and, as a consequence, their familiarity with more formal languages. The study of Kaufmann *et al.*, however, showed that the findings above apply to both categories of users likewise – to those without any prior knowledge of query languages, as well as to those with a background in software engineering and familiar with at least SQL.

4.1 Using Evolizer to answer common Program Comprehension Questions

In [9], De Alwis and Murphy have listed 36 common program comprehension questions that their tool *Ferret* implements. The questions fall into five categories: *inter-class*, *intra-class*, *inheritance*, *declarations*, and *evolution*. The questions are further assigned to one or several contexts, or what they call contributing *spheres*: The *static*-sphere relies on static program analysis, the *dynamic*-sphere uses profiling information, the *evolution*-sphere relies on software repository mining, and the *plug-in*-sphere contains declarative information specified in Eclipse plug-in manifests.

EVOLIZER supports all of their static queries *out of the box*, without having them predefined or hard-coded explicitly. Conceptually, we can also answer all the questions from the *evolution*-sphere.

[8] http://www.jfree.org/jfreechart/

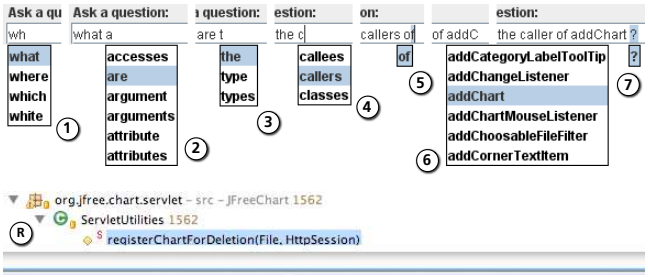

Figure 4: Entering a query and retrieving the result.

In the following we have selected, for each of the first four categories, those questions that proved to be most useful to developers in the field-study conducted by De Alwis and Murphy. We use them to exemplify how EVOLIZER can be used to support program comprehension. As a case study, we use Release 1.0.12 of JFreeChart, an open-source chart library written in Java with a size of more than 250 kLOC.

Questions concerning Inter-Class Dependencies

De Alwis and Murphy identified the question *"What calls this method?"* to be the most commonly asked one when a developer is trying to understand a system. The question falls into the category of inter-class dependencies and can be easily answered with EVOLIZER, as well as by many existing tools – including Eclipse itself. We have randomly chosen the class `ChartDeleter` from JFreeChart. The class declares four methods, among them `addChart(String)`. To find its callers, we can enter exactly:

What methods call addChart?

into the input field of Ginseng and execute the query. Figure 4 illustrates how Ginseng provides guidance for the developer to compose a query: When she starts to type *"W"*, a list of possible question words pops up (Step 1). After selecting the word *"What"*, she types *"a"* and receives several suggestions starting with that letter, such as *"accesses"*, *"are"*, *"arguments"*, and so on. Going on like that, the developer is able to compose the complete query (Steps 3 to 6) and, as soon a valid query was entered, she can execute it by concluding the sentence with a question mark (Step 7). This kind of guidance is especially valuable for novice programmers, who are not already familiar with the underlying knowledge base.

In case of JFreeChart, a single match is presented after the execution of the query: `registerChartForDeletion(File, HttpSession)` of the class `ServletUtilities` (Step R). This corresponds to the result of invoking the *"Find references in project"*-functionality of Eclipse.

Variations of the initial question are also possible:

What are the callers of addChart?

is accepted by Ginseng just as well as the imperative form:

Give me the methods invoking addChart!

We want to remark that Ginseng automatically generates these variations solely based on its grammar rules, synonyms encoded in the ontology, and instance data provided by EVOLIZER. There is no need to explicitly define in advance the questions that are possible – developers can ask them based on the facts extracted from source code.

Another program comprehension question that was identified to be asked often by developers is *"What fields are declared as being of this type?"* The structure of the original question is too complex for the simple grammar rules of Ginseng but can be reformulated to: *"What fields have the type <type>?"* For example, we can search for attributes of type `JTextField` to identify classes of JFreeChart that contribute to the user interface in general and, in particular, process user input. Entering the question:

What attributes have the type JTextField?

returns seven results; three in `DefaultAxisEditor`, two each in `DefaultNumberAxisEditor` and `DefaultTitleEditor`.

This time significantly more user interaction is necessary to come up with the same results using the *Java Search* of Eclipse. Besides entering the search string, we have to choose *Search for type* and configure the *"Limit To"* option to only match for *field types*. Especially newcomers to Eclipse are often not aware of these features.

Questions concerning Intra-Class Dependencies

During maintenance tasks, developers often face god-classes several thousand lines of code in size. Changing, *e.g.,* the type of one of their attributes is a tedious task, involving careful code investigation and a lot of scrolling to answer questions, such as *"What methods access this field?"* Often, the task is further complicated when information hiding principles were violated. Using our framework, we are able to significantly narrow down the amount of code that has to be inspected. Coming back to one of the examples in the last section, we can ask what methods access the field `labelFontField` in `DefaultAxisEditor`. Again, the user is guided during query composition. When she starts typing *"What method accesses..."*, Ginseng automatically suggests *attribute* (as well as field, as a synonym) and *method* as the next possible words. By choosing *attribute*, the set of suggested candidates for the concluding word is reduced to the names of particular fields, *i.e.,* names of methods are faded out. The full query is:

What method accesses the attribute labelFontField?

Executing the query yields two results: the constructor of `DefaultAxisEditor` and the method `attemptLabelFontSelection()`. Manual inspection confirms these results to be correct.

Questions concerning Inheritance

Generalization and specialization are among the most powerful features in Object-Orientation. The other side of the coin is that inheritance increases the gap between the static program and the dynamic process and therefore complicates program understanding in some cases, especially if used too extensive (*e.g.,* deep inheritance hierarchies over more than seven levels), or incorrectly (*e.g.,* to simply reuse code, rather than backed by the idea of specialization). Supporting the developer with tools to understand inheritance hierarchies more easily is therefore desirable. Browsing through code and navigating upwards the inheritance hierarchy is already well-supported by modern IDEs. Navigating downwards, on the other side, usually involves tedious searching. However, literally speaking, it could be as easy as asking *"What are the subclasses of this class?"*, if our approach were used.

The class `DefaultAxisEditor` in JFreeChart is implemented as a subclass of `JPanel`. Querying the static source code information in FAMIX, we can quickly locate similar classes, *i.e.,* classes that extend `JPanel`: `DefaultTitleEditor`, `DefaultAxisEditor`, and `FontChooserPanel`, among others. If the underlying ontology provides meaningful synonyms, each of the following queries would return the information that we are interested in:

- *What are the classes that extend JPanel?*
- *What are the subclasses of JPanel?*
- *What classes inherit from JPanel?*

Questions concerning Declarations

We have chosen *"What are all the fields declared by this type?"* among the 36 conceptional queries that De Alwis and Murphy have listed in their paper to conclude our case study on how developers can benefit from our framework. The question has been identified by De Alwis and Murphy to be the most commonly asked one concerning declarations. In the last example, we have identified a few classes that are subclasses of `JPanel`. If we want to confirm the initial impression that `DefaultTitleEditor` and `DefaultAxisEditor` implement similar concepts according to their naming scheme, we can do that in terms of comparing their states, *i.e.,* fields. Investigating the answers to the queries:

What attributes are defined in DefaultAxisEditor?

and:

What are the attributes of DefaultTitleEditor?

confirms that they, in fact, have similar states: For example, both of them seem to have associated a font (`labelFont` and `titleFont`, respectively), a checkbox (`showTickLabelsCheckBox` and `showTitleCheckBox`, respectively), and so on. The next steps would probably be to investigate further the types of the fields and the behavior that operates on them or to have a look at the documentation of the two classes, *e.g.,* by entering the following query:

Give me the Javadoc of DefaultAxisEditor!

From here, gaining a deeper understanding of JFreeChart is just a matter of asking the right questions.

4.2 Discussion and Limitations

We conducted a validation of our approach by addressing the questions that De Alwis and Murphy listed in their paper about Ferret. The two approaches share many similarities in terms of their goals. Furthermore, those questions were identified in two empirical studies [34, 33] to be the most frequently asked questions by programmers during software evolution tasks and therefore provide a suitable benchmark for our framework.

Our approach, in contrast to Ferret, can draw from the full power of the semantic web technologies and is therefore not limited to a set of predefined queries, with the need to have any additional ones implemented by some provider, such as a tools-support group. Instead, the querying capabilities of our framework are much more flexible and only limited by a subset of the English grammar and by the knowledge base that is available. Therefore the developer queries that we have chosen in our case study are only a subset of the ones that can be formulated and answered *out of the box*. Many more are possible and they can be formulated in different variations, *e.g.,* as a questions or using the imperative form.

IDE vendors need to provide an interface to the information offered by tools comparable to Ferret (often menu-items in deeply nested context-menus). This becomes more and more of a problem as the information need of developers may become more diverse with the increase in complexity of modern software systems. Our framework, in contrast, provides a single access point for most of the information needs: using natural language, a developer can just ask what is on her mind, without having to worry where the desired functionality is hidden in the deeply nested menus of her favorite IDE.

Existing query languages for software evolution artifacts potentially also provide such an access point and give the developer the freedom to formulate queries without being bound to a set of predefined ones. On the other hand, they usually rely on custom-tailored, verbose languages. Therefore, they are hardly used in practice. A simple question, such as *"What methods call addChart?"*, which our tool answers right away, has to be reformulated by a developer into a SQL-like statement in order to be answerable with a tool, such as Semmle.[9] A Semmle query would look like this:

```
from Method caller, Method callee
where caller.calls(callee)
and   caller.fromSource()
and   callee.fromSource()
and   caller != callee
and   callee.getName().matches("addChart%")
select caller.getName()
```

Moreover, extending existing query languages with new vocabulary involves manual editing of the language definition files, whereas in our framework, additional vocabulary is available as soon as new data is loaded into EVOLIZER. This is possible because Ginseng generates dynamic grammar rules from the loaded OWL ontologies, but it also implies that we have to rely on meaningful identifiers in the ontologies that we query. If this is not the case, we also have to fall back to manual definitions of synonyms. This is straight-forward and can be done either in advance by a tool-vendor or later by the end-users, *i.e.,* developers – even if they are unfamiliar with the Semantic Web – in case that they are more comfortable with another vocabulary than the one that is already provided by the ontology.

Query languages are less ambiguous than natural language in general and therefore better in expressing, *e.g.,* complex restrictions and in formulating composed queries. However, supported by the empirical studies mentioned above, we claim that the most common program comprehension questions have a simple structure. In rare cases where more expressive power is needed, one can always fall back to SPARQL or to the SQL-like Hibernate Query Language (HQL).

The performance of our prototype on a common laptop computer is acceptable for a project of the size of JFreeChart (∼250kLOC, the response time for the queries presented in our case study was usually around a couple of seconds).

5. RELATED WORK

Our work is highly related to the approach of De Alwis and Murphy presented in [9]. Just like them, we offer a framework to support the composition and integration of different sources of data about software artifacts in a single queryable knowledge base. In contrast to our approach, they define their own *sphere model*, whereas we rely on standardized technologies that are already established in the research community, as well as in industry. Moreover, while Ferret restricts developers to a set of predefined, hard-coded questions, we give them the freedom to formulate their own questions by exploiting existing natural language query tools.

Natural Language in Program Comprehension

LaSSIE, presented by Devanbu *et al.* in [11], integrated multiple views on a software system at AT&T in a frame-based knowledge base and also provided semantic retrieval through a natural language interface. LaSSIE and our framework share many common-

[9] http://www.semmle.com/

alities, especially since the Semantic Web emerged from frame-based knowledge representation techniques. Hill *et al.* presented an algorithm to extract noun, verb, and prepositional phrases from method and field signatures in source code to enable *contextual searching* [21]. The queries they support are closer to keyword search on identifiers found in source code than to full natural language questions and they do not cover structural information, such as caller-callee or inheritance relationships among source code entities.

Query Languages for Software Artifacts

Many approaches have been proposed that use specific languages to query software artifacts. They are either based on standard database languages, such as SQL or Datalog (*e.g.,* CodeQuest [17] and Semmle), customized Prolog implementations (*e.g.,* JQuery [23], AST-Log [8], GraphLog [7]), or a custom language (*e.g.,* SCA [29]). All of them aim to help developers to effectively explore and better understand code, uncovering information that would be impossible or extremely hard to find with standard tools. However, most of them require the user to master syntax and vocabulary of a specific query language limited to that single purpose. Our approach guides developers in vocabulary, as well as in syntax, to construct well-formed and coherent questions about static source code information. Nevertheless, we consider most of these query languages complementary to our approach, as they are more expressive in terms of that it is possible to compose more complex queries than with the subset of English grammar rules that we rely on. In general, as argued in Section 4, the most common questions that arise during software evolution tasks are of simple structure and are therefore predestinated to be answered with natural language using EVOLIZER.

Semantic Web in Software Engineering

Our framework relies heavily on Semantic Web technologies. Besides the Web, these technologies have proven to be useful in many domains, for example to enable the interoperability of software systems, and when technologies are needed to express knowledge with formal semantics to enable machine processing. Software Engineering is one of these domains. An overview of applications of ontologies in software engineering has been given in [19, 16, 36]. All of these publications promote the theoretical benefits offered by different characteristics of ontologies, such as explicit semantics and taxonomy, lack of polysemy, ease of communication and automatic data exchange between distinct tools, and computational inference. On the other hand, only few approaches put those ideas into real practice. Hyland-Wood *et al.* [22] propose an OWL ontology of software engineering concepts (SEC), including classes, tests, metrics, and requirements. Bertoa *et al.* [4] follow a similar approach but focus more on software measurement. Happel *et al.* [18] propose various ontologies to foster software reuse. In their KOntoR approach, they provide therefore background knowledge about software artifacts, such as the programming language and licensing models. Kiefer *et al.* developed EvoOnt, a software repository data exchange format based on OWL [25]. Their software ontology model (SOM) was influenced by FAMIX. Their version ontology model (VOM) and their bug ontology model (BOM) are based on EVOLIZER's data models for CVS and bug tracking information, respectively. The authors use iSPARQL, their extension to the RDF query language SPARQL, for effectively querying their ontologies to detect code smells. Witte *et al.* [37] use text mining and static code analysis to map documentation to source code for software maintenance purposes. These mappings are represented in RDF. The MOST project [27] aims to facilitate software engineering by leveraging ontology and reasoning technologies. It integrates ontologies into model-driven software development (MDSD), resulting in ontology driven software development (ODSD). All of the approaches mentioned above acknowledge the potential of ontologies and the Semantic Web applied to software engineering. They often define custom ontologies that can be integrated in the ontology layer of EVOLIZER.

6. CONCLUSIONS

As software systems get more complex, efficient tools to support software engineers during their development and maintenance tasks are becoming more important. Modern IDEs already made a great leap forward in providing a variety of features to, for example, facilitate program comprehension. The complexity of the user interface is putting a significant cognitive burden on a developer. Often it is easier to solve a task manually than to master a tool. Although experienced developers usually know exactly what information they are looking for, they often do not know how to get it. They simply do not know how to turn conceptual queries into something their IDE understands.

In this paper, we presented a framework that overcomes this gap and showed an application of Semantic Web technologies that goes beyond merely data exchange for the sake of tool interoperability. We combined industrial-strength technologies with ideas and tools from the Semantic Web to enable developers to query software engineering artifacts in a way that is familiar to them: using (quasi) natural language strongly resembling plain English. For that, we use the Web Ontology Language OWL to describe static source code information extracted by our EVOLIZER. The resulting ontology then serves as input for the guided-input natural language interface Ginseng. We demonstrated in a case study that our approach makes it possible to answer the most common program comprehension questions identified in the literature.

We do not restrict developers to a set of predefined questions but advance the state-of-the-art in that our approach is only dependent on what data is available as input. With our framework, it is straight-forward to integrate almost any kind of evolutionary information, for example, from version control or issue tracking systems – solely by exploiting existing and well-established standards for resource description. We encourage other researchers to download and try out our EVOLIZER toolset or to incorporate our SEON ontology into their own tools.

7. ACKNOWLEDGEMENTS

This work was supported by the Hasler Foundation, Switzerland as part of the *ProMedServices* project and the Swiss National Science Foundation as part of the *Enterprise Computing* and *diCoSA* projects. We also thank Esther Kaufmann for providing us Ginseng.

8. REFERENCES

[1] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396 - uniform resource identifiers (URI). IETF RFC, August 1998. `http://www.ietf.org/rfc/rfc2396.txt`.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific America*, 284(5):34–43, 2001.

[3] A. Bernstein, E. Kaufmann, C. Kaiser, and C. Kiefer. Ginseng: A Guided Input Natural Language Search Engine for Querying Ontologies. In *Jena User Conf.*, May 2006.

[4] M. Bertoa, A. Vallecillo, and F. Garcia. An ontology for software measurement. In *Ontologies for Software Engineering and Software Technology*. Springer, 2006.

[5] J. Bevan, J. E. James Whitehead, S. Kim, and M. W. Godfrey. Facilitating software evolution research with Kenyon. In *Proc. Joint European Softw. Eng. Conf. and ACM SIGSOFT Symposium on the Foundations of Softw. Eng.*, pages 177–186. ACM, September 2005.

[6] G. G. Chowdhury. *Introduction to Modern Information Retrieval*. Facet, London, 2nd edition, 2004.

[7] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proc. Int'l Conf. Softw. Eng.*, pages 138–156, New York, NY, USA, 1992. ACM.

[8] R. F. Crew. Astlog: A language for examining abstract syntax trees. In *In Proc. USENIX Conf. Domain-Specific Languages*, pages 229–242, 1997.

[9] B. de Alwis and G. C. Murphy. Answering conceptual queries with ferret. In *Proc. Int'l Conf. Softw. Eng.*, pages 21–30, New York, NY, USA, 2008. ACM.

[10] M. Dean and G. S. eds. *OWL Web Ontology Language Reference*. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/owl-ref/`.

[11] P. Devanbu, R. Brachman, and P. G. Selfridge. Lassie: a knowledge-based software information system. *Commun. ACM*, 34(5):34–49, 1991.

[12] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int'l Conf. Software Maintenance*, pages 23–32. IEEE Computer Society, September 2003.

[13] B. Fluri, M. Wuersch, M. Pinzger, and H. C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[14] H. C. Gall, B. Fluri, and M. Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Softw.*, 26(1):26–33, January/February 2009.

[15] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199 – 220, 1993.

[16] M. Gruninger and J. Lee. Ontology applications and design. *Communications of the ACM*, 45(2):39–41, 2002.

[17] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *Proc. European Conf. Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[18] H. Happel, A. Korthaus, S. Seedorf, and P. Tomczyk. KOntoR: An Ontology-enabled Approach to Software Reuse. *Proc. Int. Conf. Softw. Eng. and Knowledge Eng.*, 2006.

[19] H.-J. Happel and S. Seedorf. Applications of ontologies in software engineering. In *Workshop Semantic Web Enabled Soft. Eng. at Int'l Semantic Web Conf.*, Galway, Ireland, November 11-15 2006.

[20] S. Henninger. Using iterative refinement to find reusable software. *IEEE Softw.*, 11(5):48–59, 1994.

[21] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proc. Int'l Conf. Softw. Eng.*, pages 232–242, Washington, DC, USA, 2009. IEEE Computer Society.

[22] D. Hyland-Wood, D. Carrington, and S. Kaplan. Toward a Software Maintenance Methodology using Semantic Web Techniques. *Proc. Int'l IEEE Workshop on Softw. Evolvability at IEEE Int'l Conf. on Softw. Maintenance*, pages 23–30, 2006.

[23] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. Int'l Conf. Aspect-Oriented Softw. Development*, pages 178–187. ACM, 2003.

[24] E. Kaufmann and A. Bernstein. How Useful are Natural Language Interfaces to the Semantic Web for Casual End-users? In *Int'l Semantic Web Conf.*, pages 281–294, March 2007.

[25] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with isparql and a software evolution ontology. In *Proc. Int'l Workshop on Mining Software Repositories*, page 10, Washington, DC, USA, 2007. IEEE Computer Society.

[26] G. Klyne and J. J. C. eds. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[27] MOST (Marrying Ontology and Software Technology) Project homepage, Last visited October 2008. `http://www.most-project.eu/`.

[28] P. F. Patel-Schneider, P. Hayes, and I. H. eds. *OWL Web Ontology Language Semantics and Abstract Syntax*. W3C Recommendation, 10 February 2004. `http://www.w3.org/TR/owl-semantics/`.

[29] S. Paul and A. Prakash. A query algebra for program databases. *IEEE Trans. Softw. Eng.*, 22(3):202–217, 1996.

[30] M. Pinzger, E. Giger, and H. C. Gall. Handling unresolved method bindings in eclipse. Technical report, Department of Informatics, University of Zurich, Switzerland, 2007.

[31] M. Pinzger, K. Gräfenhain, P. Knab, and H. C. Gall. A Tool for Visual Understanding of Source Code Dependencies. In *Proc. Int'l Conf. Program Comprehension*, pages 254–259. IEEE Computer Society, 2008.

[32] E. Prud'hommeaux and A. S. eds. SPARQL query language for RDF. W3C Recommendation, 15 Jannuary 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[33] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *Proc. ACM SIGSOFT Symposium Foundations of Softw. Eng.*, pages 23–34, New York, NY, USA, 2006. ACM.

[34] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2008.

[35] S. Tichelaar, S. Ducasse, and S. Demeyer. Famix and xmi. In *Proc. Working Conf. Reverse Eng.*, page 296, Washington, DC, USA, 2000. IEEE Computer Society.

[36] M. Uschold and R. Jasper. A framework for understanding and classifying ontology applications. In *Proc. IJCAI Workshop Ontologies and Problem Solving Methods*, August 1996.

[37] R. Witte, Y. Zhang, and J. Rilling. Empowering software maintainers with semantic web technologies. In *European Semantic Web Conf.*, Innsbruck, Austria, 3-7, June 2007. Springer Verlag.

[38] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Int'l Workshop Mining Software Repositories*, May 2004.

[39] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, June 2005.