SmartAPI - Associating Ontologies and APIs for Rapid Application Development

Andreas Eberhart Sudhir Agarwal

AIFB, University of Karlsruhe
{aeb,agarwal}@aifb.uni-karlsruhe.de
http://www.aifb.uni-karlsruhe.de/WBS

Abstract

Software developers are confronted with an ever-increasing number of Application Programming Interfaces (APIs) for various areas such as IO, database connectivity, XML, regular expressions, and many others. Learning to use these interfaces is quite difficult and time consuming. At the same time, the very structured nature of programming languages lends itself to being represented in an ontology. We pick up this idea and present the SmartAPI project, which semantically tags APIs in order to support advanced searching capabilities and automatically suggesting how API calls can be chained together. We also show, how SmartAPI can be deployed seamlessly via an editor plug-in and a Web Service based backend. Future versions will incorporate Web Services and model side effects as well.

1 Introduction

Software developers are confronted with an ever-increasing number of Application Programming Interfaces (APIs) for various areas such as IO, database connectivity, XML, regular expressions, and many others. The use of standard design patterns within these APIs is a big help, but especially beginners frequently struggle when having to use an unfamiliar API. An experienced C programmer will quickly understand the concepts of a modern language such as C# or Java. However, learning how to use a massive API still takes a lot of time.

Consider the following scenario: The programmer needs to read data from a database via the JDBC interface. The system administrator of the company provided user name and password, which obviously need to be used in the process. We believe that an integrated development environment (IDE) should be able to assist the programmer in the following two ways:

1. Search the entire API for a method call, which takes a database user name as an input parameter.

2. Suggest, how various API calls should be sequenced in order to go from the connection information all the way to actually receiving data from the database.

We believe that the reason why the aforementioned features are not supported by the available IDEs is that the APIs are not semantically rich. They contain syntactic information, which the programmers have to read and interpret, which makes understanding, learning and using an API a very time consuming task. However, we believe that such features can be realized by applying ideas from the areas of "Knowledge Management" and "Knowledge Representation". The enrichment of purely syntactic information of APIs with semantic information will allow the computer to perform certain tasks that normally the human programmer has to perform.

The rest of the paper is organized as follows: The next section discusses the structure and information content of today's APIs and explains which added value an ontology can provide. Section 3 describes the algorithms we use in order to implement the suggested functionality. Section 4 outlines the actual system architecture before we conclude the paper by summarizing the results and outlining future research directions.

2 Ontologies for Application Programming Interfaces

We employ the new technologies and languages provided in the Semantic Web [2] for representing the required knowledge. Ontologies play a very prominent role within this community. According to Gruber, an ontology is "a formal, explicit specification of a shared conceptualization" [4]. Consequently, ontologies promise to not only be a useful vehicle for knowledge representation, but also provide meaningful terms, the developer can understand and interpret correctly.

The first step when engineering an ontology is to establish a concept taxonomy. Guarino and Welty established some formal guidelines for checking the correctness of such a taxonomy [5]. For example, human cannot be a subclass of species, since a human is identified by a position in space and time. Formally, concept meta attributes are defined which are inherited by the child concepts. Errors manifest themselves by conflicts with inherited attributes and the ontology engineer's interpretation of the concept. It turns out that programming APIs exhibit a very clean structure. Here, flaws in the class hierarchy would result in a poorly designed API. Especially very common APIs such as the Java API from Sun are very well designed. Consequently, one can argue that an API already represents the taxonomic backbone of an ontology.

The inter-object relationships are encoded in the methods and therefore not clearly exposed to the user. Also not every concept is explicitly modelled in the

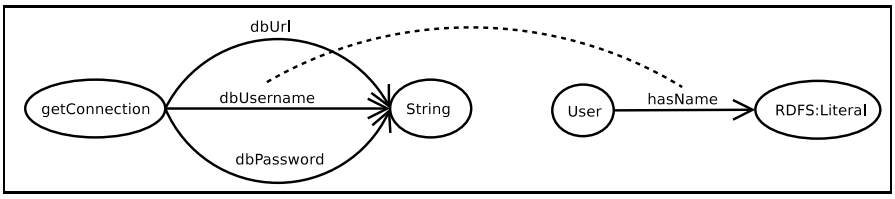

Figure 1: Enriching APIs with Ontology Markups

API. Primitive and simple data types such as numbers and strings are used for representing a range of very different concepts. For example URLs, passwords, relative file paths are all treated as strings in the various programming languages. Currently, programmers have to know the semantics of such concepts, which makes it difficult to understand and use a new API. We believe that this can be improved by enriching APIs with semantical information.

Classes such as `java.sql.ResultSet` already carry a lot of information. Information about a class can be automatically converted into an ontology by using Java reflection API[1]. Doing such a conversion makes it possible to relate the concepts and properties to other concepts and properties without having to program a single line of code. By specifying such associations one gives the concepts and properties (especially those of primitive types) more meaning.

In our earlier example from section 1, the correct method to call would be `java.sql.DriverManager.getConnection(dbUrl, dbUsername, dbPassword)`. However, all the input parameters are to of type string and thus lack any further semantics. We associate these parameters with the respective ontological entities. Refer to the dashed curve representing one such association (between `dbUsername` and `hasName`) in figure 1. Specificying such associations is not difficult and enables semantical search for methods of an API.

3 SmartAPI

This section explains the methodologies and algorithms employed for the implementation of the search and sequencing features described above. We will first describe the features in more detail.

In such a setting, a user will find the method `getConnection` if he searches for the methods that have a database user name as a parameter much more easily than if he performs a search that is based on the syntactical information, i.e. searches for the methods that have parameters of type string. This is only possible, since the implicit knowledge of the method `getConnection(String dbUrl, String dbUsername, String dbPassword)` (in this case hidden in the variable names) is now available in the ontology in machine understandable format.

The second use case is the chaining of method calls which has been suggested

[1] http://java.sun.com/docs/books/tutorial/reflect/index.html

by the authors before in the context of Web Services [1, 3]. Assume the user has local variables or parameters with the database URL, username, password, desired SQL query, and desired column name available. The goal is to connect to the database using the first three pieces of information, and to read data using the last two. If the user associates the variables with the respective ontological terms, SmartAPI is capable of automatically establishing the following calling sequence.

```
try {
    Connection connection =
       DriverManager.getConnection(dbUrl, dbUsername, dbPassword);
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery(dbSelectQuery);
    String dbTableValue = resultSet.getString(dbColumnName);
}
catch (SQLException e) {
    e.printStackTrace();
}
```

This piece of code is computed using a breadth first search algorithm. The graph vertices to be searched are sets of variables available at a certain point of the execution. The starting state comprises the tagged variables provided by the user. A connection from vertex A to vertex B exists, if there is a method call which can be performed using the variables available in state A. The newly obtained return value combined with the previously available variables of A must then be equal to the variables in B. Assume there is a method $C_b f(C_a)$, A has variable a of type C_a, and B has variables a and b of types C_a and C_b. Then, we can get from A to B by calling $f(a)$.

SmartAPI aborts the search if no solution is found in the specified time constraint or once the first path has been found. Our experience shows that the shortest path (which will be found first in breath first search) is usually the best guess.

Static methods, non-static methods, and constructors are all handled using the same program logic. Our class `SMethod` contains the respective classes `Method` and `Constructor` of the reflection package and encapsulates the rest of the search logic from the specifics of these individual programming constructs.

The suggested code above exposes one of the current weaknesses of SmartAPI. While this code skeleton will be a big help, some things are missing. First, the driver class needs to be loaded in the beginning via `Class.forName(driver)`. Secondly, when the data is read in the last line of the try block, the cursor needs to be moved forward within a while loop using `while (res.next())`. SmartAPI ignores these calls, since they do not return any new values. They rather have a side effect on the cursor state and the Java class loader, which is not captured at the moment. Future versions of SmartAPI will address this problem.

4 Architecture

The planned architecture of the system is illustrated in figure 2. We use the KAON[2] editor to create and maintain the ontology [6]. Note that KAON also supports collaborative engineering. The Java API structure is read in a fully automatic fashion via the reflection API. The association tool is then used to establish cross-links between the shared ontology and the API information. This composite ontology is made available to both development client and SmartAPI server. Technically, we only ship the ontology file along with the associations, since the API information can be generated on any Java VM.

The SmartAPI server provides two Web Services for searching and composing method calls. The required inputs need to be gathered from the user. Consequently, it makes most sense, to embed the required forms in a Java IDE. Eclipse[3] is a good choice here, since it is a very popular editor, which offers a convenient and open plug-in framework.

Deploying SmartAPI in a client server setting makes sense, since ideally the ontology should be developed as a joint open knowledge initiative with the results being available to the public. Consequently, a user can simply download the plug-in and benefit immediately without having to worry about ontological issues at all.

We are currently working on the client-server setting as well as the eclipse plug-in. At the moment, inputs and results need to be transferred via copy and paste between the IDE and the standalone SmartAPI GUI written using Swing. However, the core system with the association tool, standalone GUI, and search engine is operational[4].

5 Conclusion and Future Work

In this paper we showed how a programmer can benefit from a semantically enriched API. The user can search a suitable class or method using common terms from the ontology. It is even possible to compute a suitable sequence of method calls, given a starting set of variables and the desired result. We explained the relationship between ontologies and APIs and how to combine the two. Furthermore, we devised an algorithm for representing variable state changes in software and developed a search algorithm, which results in a sequence of calls. The system also contains a rudimentary user front-end. Future work was partly already shown in the architecture section. Another major point is a deeper description of methods in order to capture side effects as well as extending this paradigm to Web Services. The authors already conducted some initial research on these issues [3]. Finally, we plan to evaluate the users' benefit and acceptance of SmartAPI.

[2] http://kaon.semanticweb.org

[3] http://www.eclipse.org

[4] Screenshots of the tools are available at http://www.aifb.uni-karlsruhe.de/WBS/aeb/smartapi/. We plan to add a download soon as well.

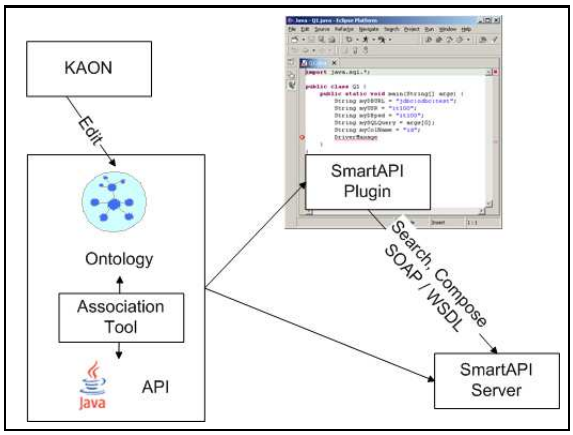

Figure 2: Planned client-server architecture. The client is deployed as an eclipse plugin which connects to the SmartAPI server via SOAP and WSDL.

References

[1] S. Agarwal, S. Handschuh, and S. Staab. Surfing the service web. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *Second International Semantic Web Conference*, volume 2870 of *LNCS*, pages 211–226. Springer, OCT 2003.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, pages 28–37, May 2001.

[3] A. Eberhart. Towards universal Web Service clients. In B. Hopgood, B. Matthews, and M. Wilson, editors, *Proceedings of the Euroweb 2002: The Web and the GRID: from e-science to e-business*, Oxford, UK, December 2002. http://www1.bcs.org.uk/DocsRepository/03700/3780/eberhart.htm.

[4] T. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, NL, 1993. Kluwer Academic Publishers.

[5] N. Guarino and C. Welty. Evaluating ontological decisions with ontoclean. *Communications of the ACM*, 45(2), Feburary 2002.

[6] A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. Ontologies for enterprise knowledge management. *IEEE Intelligent Systems*, 18(2):26–33, March/April 2003.