A SEMANTIC SEARCH ENGINE FOR SOFTWARE COMPONENTS

Bernhard G. Humm
Hochschule Darmstadt – University of Applied Sciences
Haardtring 100, 64295 Darmstadt, Germany
bernhard.humm@h-da.de

Hesam Ossanloo
Object ECM GmbH
Grimm 3, 20457 Hamburg, Germany
hesam.ossanloo@object.ch

ABSTRACT
Software development today means, to a large extent, integrating existing software components. An important task of the architect of a software solution is to identify suitable software components. Whereas semantic search engines have gained popularity in the last decade, semantic search engines for finding software components are not yet in widespread use. The paper presents the concept for a semantic search engine for software components called “SoftwareFinder”. SoftwareFinder uses a simple ontology for normalizing the tag sets of various software hosting sites, for providing a semantic AutoSuggest service and a semantic faceted search, and for recommending similar software components. SoftwareFinder has been implemented prototypically and exhibits a good user experience as well as a good run-time performance.

KEYWORDS
Semantic search, ontology, software components

1. INTRODUCTION
Software development today means, to a large extent, integrating existing software components. A component is a well-defined unit of software that has a published interface and can be used in conjunction with other components to form larger units (Hopkins, 2000): libraries, frameworks, web services, and entire applications. Examples are database management systems, middleware components, and GUI libraries. An important task of the architect of a software solution is to identify suitable software components – commercial, free or open source. An application which takes advantage of reusable components has been argued to be more reliable (Vitharana & Jain, 2000), increase productivity of developers (Pearson, 1999) (Lim, 1994), reduce skill requirements (Kiely, 1998), shorten the development lifecycle (Due, 2000), reduce time-to-market (Kim & Stohr, 1998) (Patrizio, 2000), increase quality (Sprott, 2000), and decrease development costs (Due, 2000) (Patrizio, 2000).

As the number of components grows rapidly, a major problem in software reuse is the lack of efficient means to search and retrieve reusable components (Spinellis & Raptis, 2000). Usually, software architects rely on software components they have had experience with. Faced with new problem domains, they are left alone to asking colleagues or consulting general-purpose search engines like Google. Alternatively, they consult various sites for hosting software components like GitHub, Apache.org or Sourceforge.net.

Wouldn’t it be advantageous to use a semantic search engine for software components? For example, by asking “free Java library for machine learning” it will return a list of suitable software components like Weka and RapidMiner.

Semantic search engines have gained popularity in the last decade (Sirisha, et al., 2015). They have been established in a variety of application domains, including hotel portals, patent retrieval, dating sites etc.
Surprisingly enough, semantic search engines in the computer science field, and in particular for software components, are not yet in widespread use.

This paper presents the concept for a semantic search engine for software components which we call “SoftwareFinder”, as well as a prototypical implementation. The remainder of this paper is structured as follows. Section 2 details the problem by means of requirements. Sections 3, 4 and 5 are the core of the paper presenting the concept, system architecture, and prototypical implementation of SoftwareFinder. Section 6 evaluates our approach. Section 7 compares related work. Section 8 concludes the paper and outlines future work.

2. PROBLEM STATEMENT

In this section, we detail the problem by means of end-users’ and developers’ requirements for the semantic search engine SoftwareFinder.

2.1 End-Users’ Requirements

1. SoftwareFinder shall facilitate software architects finding relevant software components for a particular need.
2. SoftwareFinder shall include information about a sufficiently large set of software components so that for most cases, software architects will not need to consult other information sources.
3. SoftwareFinder shall provide best user experience. This shall include at least the following features and properties:
   a. An easy-to-use search box as known from popular search engines like Google.
   b. A relevance-based ranking of search results.
   c. Semantic search facilities based on an ontology for software components. The ontology shall unify the terminology used in various software hosting sites for describing software components (“tags”). It shall be sufficiently comprehensive, thus facilitating uniform search and comparison of software components hosted on different sites. The ontology shall at least contain the following information: preferred tag label, category, synonyms, and acronyms.
   d. A semantic AutoSuggest service that supports users in formulating search terms, based on the ontology.
   e. A faceted semantic search service that allows users to refine and narrow down search results, based on the ontology.
   f. An overview of the search history, which allows navigating in the search space by modifying individual search terms.
   g. Provision of sufficient information about a selected software component, including description, semantic tags, and links to project website and download site.
   h. A facility for recommending semantically similar software components of interest compared with a particular software component selected.
   i. Responsive design, i.e., ability for using SoftwareFinder on various electronic devices in a user-friendly way.
   j. Response time of less than 1 s for most searches.

2.2 Developers’ Requirements

1. The ontology shall be easily extensible.
2. New relevant software hosting sites can be integrated with moderate effort.
3. SOFTWAREFINDER: A SEMANTIC SEARCH ENGINE FOR SOFTWARE COMPONENTS

In this section, we describe the concept of the SoftwareFinder application by means of the example “free Java library for machine learning”. While the user starts typing “machine learning” in the search box, the most relevant matching search terms are suggested to the user. See Figure 1.

All suggested terms are semantically categorized, according to the ontology. In total, 16 categories are used, including “development”, “communication”, “business”, “engineering”, “humanities”, and “science”. The categories have been specifically designed for the SoftwareFinder application using classification techniques as in (Lucredio, et al., 2004).

The user may, of course, also use any other search term as well as wildcards, e.g., “mach*”.

After choosing “Machine learning” from the AutoSuggest service, a list of software components is displayed which contain the tag “Machine learning”. The list is sorted according to the rating of the software components. The rating (1 star to 5 stars) is a normalization of different rating schemes of the various hosting sites.

In order to narrow down the search result, there is an interaction element which is called “topic pie” (Deuschel, et al., 2014) (Figure 2). In the topic pie, relevant search terms are displayed on the outer ring. The semantic categories of the terms are displayed in the inner ring and discriminated using a color coding. By selecting a search term, the search will be refined, resulting in a smaller search result set. Also, the topic pie will be adapted accordingly.
The terms displayed in the topic pie are identified according to a heuristic approach balancing relevance and diversity. For more details, refer to (Deuschel, et al., 2014). Semantic AutoSuggest and topic pie support users in specifying search terms particularly in areas where they are novices and do not yet know the expert vocabulary.

The back button in the navigation bar allows the user to go one step backwards in his search history. Additionally, the navigation bar enables the user to delete or replace one of the search terms thus allowing for exploring the search space.

If a software component is selected, a detail page is displayed (Figure 3). On the detail page, users find a short description of the software component, subjects (a.k.a. tags or keywords), programming language, license type and license details, rating, the home page of the component and, the link to download the component.

Additionally, similar software components are displayed. This feature fosters serendipity and is inspired by the option in e-commerce applications: “people who bought this product also bought...”.

SoftwareFinder has a responsive design which adapts according to the resolution of the device being used. See Figure 4 for screenshots of the same application state on a smartphone, a tablet, and a desktop. The smartphone offers the smallest display size. Therefore, the topic pie is minimized. By clicking on the arrow on the right bottom side, the topic pie will be enlarged and can be used as in the tablet or desktop version.

4. SOFTWARE ARCHITECTURE

Figure 5 gives an overview of the software architecture of SoftwareFinder. The software architecture is separated into an online and an offline subsystem. The offline subsystem manages the crawling of the software hosting sites, regularly updating the SoftwareFinder data store as a batch process. The online subsystem performs the semantic search.

4.1 The SoftwareFinder Offline Subsystem

The SoftwareFinder offline subsystem follows the software pipeline architectural style. The crawler visits software hosting sites and saves the HTML pages for the individual software components hosted. Afterwards, a semantic ETL (Extract, Transform, Load) process starts extracting metadata of the software components from the HTML pages, transferring them into a uniform format, preprocessing them semantically and loading them into the data store.

Figure 5. SoftwareFinder architecture

Figure 6. Tag normalization
Tag Normalization

The ontology is used for automatically normalizing tags of software components from different hosting sites (Figure 6). For example, the synonym tags “Monitor” and “Monitors” are unified to the preferred tag “Monitor”. Acronyms are handled, e.g., “CMS” is replaced by “Content Management System”. Blacklisted tags such as “Other/Nonlisted Topic” are omitted and compound tags such as “Project and Site Management” are split up.

4.2 The SoftwareFinder Online Subsystem

The online subsystem is designed as a classical three-layer-architecture, consisting of client, semantic logic, and data store. The data store contains metadata about software components and the ontology. It is indexed for high-performance access. The client implements the SoftwareFinder GUI including the responsive design. It accesses the semantic logic via an API. The semantic logic covers the various aspects: semantic AutoSuggest, topic pie, and similar software components.

Semantic AutoSuggest

For the semantic AutoSuggest service, all tags in the ontology are indexed, as well as all programming languages, operating systems, license types, and software product names. Initially, the tags matching a user input are ordered according to a heuristics-based relevance ranking. The heuristic used is as follows: the more often a tag is used in the software product metadata, the higher its relevance.

Only the top 7 tags out of potentially hundreds of matching tags will be displayed to the user. Using the relevance-based ranking only has a disadvantage: in many cases, tags of just one category will be displayed. In order to increase the category diversity, the initial, solely relevance-based ranking result is reordered. By omitting excess terms of the same category, it is ensured that the user has tags from at least three different categories to choose from.

Topic Pie Generation

The topic pie is generated in several steps (Figure 7).

Input to the topic pie generator is the result set of the current search, i.e., a list of software product metadata. First, the tags are extracted from the metadata. The extracted tags are then grouped according to their categories and ordered using a heuristics-based relevance ranking. The heuristic used is: the more often a tag is used in the result set, the more relevant it is. So, the rank of an individual tag is the number of its occurrences in the result set. The categories are ranked as well, based on the ranks of their individual tags: the more often a category is used in the result set, the more relevant it is. So, the rank of a category is the sum of the ranks of its tags.

The topic pie accommodates for up to 25 tags, out of potentially hundreds of selected tags. Selecting just the 25 top-ranked tags often results in only one or two categories being shown to the user. Therefore, as in the semantic AutoSuggest, relevance is traded with diversity in order to display a well-balanced topic pie. By
omitting excess tags of each category, it is ensured that the user has tags from potentially five different categories to choose from.

**Similar Software Components**

For displaying software components that are similar to the one currently selected by the user, a similarity metrics is used: the more tags two software components have in common, the more similar they are. Since all tags have been normalized during the semantic ETL process, issues of synonyms, acronyms, etc. need not be considered here. From the list of similar software components ordered according to the similarity metrics, the first three will be displayed to the user.

The ontology is the core of SoftwareFinder as it is used in all semantic functionalities in the offline and online subsystems.

5. **PROTOTYPE IMPLEMENTATION**

We have successfully implemented a SoftwareFinder prototype. The server is implemented in Java 8 involving a number of third-party libraries: For crawling, the library cawl4j is used; The data store is implemented with Apache Lucene. Semantic search is implemented via Lucene’s document fields. Ranking strategies are implemented using Lucene’s custom boosting. Indexing for semantic AutoSuggest uses Lucene’s suggester based on infix matches called “AnalyzingInfixSuggester”.

The client / server communication is via HTTP using JSON as data format.

The client web app is implemented in HTML5 / CSS3 / JavaScript using various JavaScript libraries: Knockout.js is used for implementing the MVVM architecture. JQuery, jQuery UI and jQuery-touchSwipe are used for widgets and the client-server communication.

The server and the web app are deployed in an Apache Tomcat servlet container.

6. **EVALUATION**

In this section, we compare the SoftwareFinder concept and prototypical implementation with the requirements specified in Section 2.

6.1 End-Users’ Requirements

1. As demonstrated via the prototypical implementation, the SoftwareFinder concept facilitates software architects finding relevant software components for a particular need.

2. The current prototype implementation only includes software components from Apache.org, Sourceforge.net and Alternativeto.net. A number of important hosting sites like GitHub are not yet included. Therefore, the prototype does not yet satisfy the requirement that for most cases, software architects will not need to consult other information sources. The SoftwareFinder concept allows integrating other software hosting sites with moderate effort and the integration of prominent hosting sites like GitHub are planned. However, it will be interesting to see whether, after integration of about a dozen of the most prominent hosting sites, this requirement can, indeed, be achieved. A sound evaluation of this requirement is subject to future work.

3. SoftwareFinder has been designed and implemented with a particular focus on user experience. In particular:
   a. The search box is well-known from prominent search engines like Google and is intuitively to use as it provides a clear information structure (Petrie & Power, 2012).
   b. All search results are ranked by the ratings of the various software components on their hosting sites. Since different sites use different rating schemes and some sites use no rating scheme at all, a heuristic approach is used for harmonizing ratings.
c. A simple ontology containing about 25,000 tags is the core of SoftwareFinder. It is used for automatically normalizing the terminology of various hosting sites, thus facilitating semantic search and comparison of software components hosted on different sites.

d. The semantic AutoSuggest service based on the ontology extends the functionality of a simple, text-based autocomplete service. For a detailed discussion see (Beez, et al., 2015).

e. The topic pie provides a faceted semantic search service, based on the ontology. For a detailed discussion see (Deuschel, et al., 2014).

f. The navigation bar allows navigating in the search space by removing individual search terms and adding new ones.

g. For each selected software component, sufficient information is displayed: name, description, semantic tags, programming language, operating system, license details, ranking, link to project website and download site.

h. The facility for recommending semantically similar software components supports serendipity. A common use case is that a software architect already knows one software component implementing a certain feature and is interested in alternatives.

i. Responsive design has been implemented allowing SoftwareFinder to be used on various electronic devices in a user-friendly way.

j. We have measured the performance of the SoftwareFinder prototype implementation on a cloud-based virtual server with 4GB RAM and 2.0GHz to 3GHz GHz processor (depends on the traffic of the virtual server service). For random access requests the average response time is 400 ms, which is way below the required 1s.

6.2 Developers’ Requirements

1. It is of utmost importance that the ontology is always up-to-date. The ontology can conveniently be edited as a spreadsheet before being imported into the SoftwareFinder application. A number of utilities support the editor in maintaining the ontology. When new tags are identified while crawling software hosting sites, they are presented to the editor. Similarities to existing terms are displayed.

2. The integration of a new hosting site into SoftwareFinder requires implementing a new crawler. Due to predefined facilities in the SoftwareFinder application, this requires about 200 lines of Java code. The implementation effort is moderate.

7. RELATED WORK

During the last decade, a lot of work has been done in introducing semantics and ontologies in a number of application domains (Ege, et al., 2015). However, relatively few applications are in the application domain of software engineering. (Happel & Seedorf, 2006) describe a number of potential application areas of ontologies in software engineering, including component reuse.

(Yanes, et al., 2012) provide a theoretical and empirical evaluation of software component search engines for COTS (commercial off the shelf) products. They compare 18 search engines, some for finding software components, some general-purpose: 8 software component search engines, 9 semantic search engines, and Google as a traditional search engine. For our purpose, we chose the 6 top-ranked search engines according to (Yanes, et al., 2012) which were accessible at the time of writing this paper. We also added alternativeTo.net since it has gained popularity in recent years. The comparison with SoftwareFinder is illustrated in Table 1.

Table 1. Feature comparison between different semantic search engines

<table>
<thead>
<tr>
<th>Search engine</th>
<th>Semantic search</th>
<th>Semantic AutoSuggest</th>
<th>Faceted search</th>
<th>Similar product suggester</th>
<th>Search refinement</th>
<th>Specialized for software components</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoftwareFinder</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sensebot</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Factibites</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
(Sugumaran & Storey, 2003) describes an early semantic-based approach to component retrieval. Their approach includes natural language processing (NLP), a domain model, and an ontology for querying a component repository. One limitation of their work is the high cost for maintaining the component repository which is necessary for making sophisticated queries. The SoftwareFinder architecture, in contrast, makes little assumptions about the hosting sites being crawled. This is a pre-requisite for the scalability of our approach at moderate cost.

The concept of the topic pie has been introduced in (Deuschel, et al., 2014). It serves as a faceted search service which helps users to refine their search queries in areas where they are not familiar with the domain-specific vocabulary. The difference between both implementations is the underlying ontologies being used. In (Deuschel, et al., 2014), the GND integrated authoring file of the German National Library which is used for finding books in a library. SoftwareFinder uses a simple custom-developed ontology for software components.

The concept of Semantic AutoSuggest and its use in personalized medicine has been described in (Beez, et al., 2015). Whereas the concept of balancing relevance and diversity is used in both approaches, the specific implementation has been customized for the use case of component search.

8. CONCLUSIONS AND FUTURE WORK

“The shoemaker’s son always goes barefoot.” Whereas semantic search engines are commonplace in domains like hotel search etc., you can hardly find any in the software engineering application domain. In this paper, we have presented the concept and prototypical implementation of SoftwareFinder, a semantic search engine for software components. The core of SoftwareFinder is a simple ontology for terminology describing software products.

SoftwareFinder has the potential of substantially improving the process of identifying suitable software components for a new software solution. The efficiency of this process may be improved by finding components faster; the effectiveness may be improved by getting a better overview of a large set of software components and thus selecting a most suitable component.

A lot needs to be done. The prototype implementation needs a number of improvements, e.g., in the implementation of the heuristics-based ranking. Most importantly, prominent software hosting sites like GitHub need to be integrated. Subsequently, SoftwareFinder shall be thoroughly evaluated for precision and recall, as well as for user experience while being used in real software development projects.

REFERENCES


