SOFTWARE SEMANTIC PROVISIONING
Actually Reusing Software

Savino Sguera [a], Philippe Ombredanne [b], Armando Stellato [a] and Maria Teresa Pazienza [a]
[a]*DISP, University of Rome Tor Vergata, Italy*
[b]*Eclipse Software Foundation*

Keywords: Component provisioning, software reuse, semantic web services, component-oriented architectures.

Abstract: Delivering component-oriented architectures is a well-established trend in software engineering and development. Assessing software reuse scenarios goes much beyond the usual "build vs buy" dilemma that so often occurs in early stages of a software process: scouting, comparing, choosing and integrating the right set of components meeting project's requirements is still an ad-hoc and error-prone task, performed by developers with little or no frameworks and tools to support them. This paper describes the SSP (Software Semantic Provisioning) project, funded in its early stages by Google™ Inc., developed during the Google Summer of Code™ 2007 program, and incubated by the Eclipse Software Foundation; the project aims to provide an ontological description of the software domain to underlie a semantic web framework to support developers in scouting and provisioning software components. A prototypical RESTful semantic repository, and an Eclipse plug-in consuming the repository services have been implemented and will be discussed.

1 INTRODUCTION

Software development nowadays largely consists of adapting existing functionalities or components to perform in a new environment, and is biased towards delivering component-oriented architectures.

Component provisioning, choosing the right software libraries set, and integrating it as a whole, are tasks carried out by software developers and libraries providers alone, often with little or no help at all, and this usually lead to rewrite existing code, or more generally to cost and time overrun which might be avoided with the right techniques and methodologies to support analysis, design and implementation disciplines.

The very general concept which lies behind software collection and reuse can be observed (in terms of needs) and applied (through successful methodologies and technical solutions) at very different level of specializations. While very general frameworks for software delivery and provisioning may offer services for accessing and contributing to large library repositories, relying on dedicated metadata for organizing and retrieving the archived objects, there could be specific fields of interest where a more complex and organized description of the repository, tailored upon explicit needs and requirements which characterize the given domain, would improve the shareability of data, information and tools inside really active and participating communities.

Following previous research on provisioning and integration of software components and libraries by the ART group at University of Rome Tor Vergata, this paper describes the SSP (Software Semantic Provisioning) project, funded in its early stage by Google™ Inc., developed during the Google Summer of Code™ 2007 program (Sguera, 2007), and incubated by the Eclipse Software Foundation.

2 MAIN USE CASES AND BENEFITS

Despite the proliferation of provisioning systems and frameworks, the component *search and choice* activities are still carried out by developers with little or no help at all. Programmers are left to themselves scouting the web to find libraries and components, and no systematic approach nor thorough frameworks exists.

In the next paragraphs we will discuss some of the most representative use cases and the benefit that

Sguera S., Ombredanne P., Stellato A. and Teresa Pazienza M. (2008).
SOFTWARE SEMANTIC PROVISIONING - Actually Reusing Software.
In *Proceedings of the Third International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 185-188
DOI: 10.5220/0001763501850188

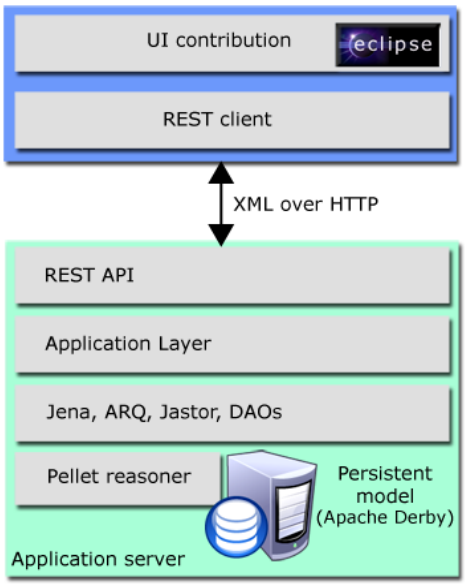

Figure 1: Full stack client-server architecture.

our approach delivers to developers and components providers, stressing how our system tackles various aspects which currently undermine software reuse and often lead to write ex-novo already existing code.

Assert and Spot Functional Equivalence between Components. The number of components and libraries, along with their versions, makes practically impossible for a developer to know them all. On the other hand, there may exist more than a piece of software accomplishing the same task, fulfilling the same requirements set, or even implementing the same specification. To some extent, such components could be considered *functionally equivalent* (at least, with respect to some facets).

This is the case, for instance, of Hibernate, Apache Cayenne and all of the other frameworks implementing the Java Persistence API, or any implementation of the Java Servlet API, any JDBC driver, or any HTTP server (or client as well). The list would go a long way…

Furthermore, the equivalence is symmetrical, reflexive and transitive; the inference mechanism helps building relations upon social-generated contents: relations and functional equivalence among software components are both explicitly declared and inferred by the system, thus building a dense semantic network with a little effort. Machine-readable metadata allow much more granularity and raise the formal level and the *intelligence* of search-related features.

Find Components Providing a Set of Tasks. Describing a software component or library in terms of the tasks it fulfills is the very first way to tell whether a piece of software fits our needs or it does not. During the analysis and design phase developers must choose the right set of enabling technologies and components which will drive further development phases, and will construct the base for building our application's architecture.

Let's suppose – just as an example – we are planning to develop two components, one carrying out the *"dom-parsing"* task and the other fulfilling the *"sax-parsing"* task, and we would like to know if there is already a unique component providing both the tasks. It would be useful to browse the repository and discover at design time that *xerces-j* actually carries out both *sax* and *dom* xml parsing. We might then decide to use it if it fits our project's requirements.

Assessing Reputation of Components. Whenever a developing team picks up third-party code to underlie its application, it is implicitly taking responsibility someone else's code, which could affect their product's security and credibility. To this purpose, we could want to know which – and how many – components actually use one: this may give us valuable information about its reputation. On the other hand, if we developed a new component – and added it to the repository – it could be interesting to know which and how many components rely on our work.

3 APPROACH AND DESIGN GOALS

Our key goal is to provide developers with a complete environment to exploit semantic metadata in order to effectively find and provision software components.

We tried to overcome the main limitations in current mainstream provisioning systems and frameworks, which are in turn tied to a particular technology or show a formalization level which grants no access to technology-independent, high level and enough granular information for a component.

Moreover, even if current provisioning technologies follow different approaches and stress different aspects proper of the software domain, there is a substantial overlap among the components' description they provide and rely upon.

Thus an ontology, meant to be a shared, higher level domain vocabulary among developers, allowing to semantically describe software and eventually mapping a subset of available metadata to one of the technologies available, would enable a thorough description of a component, aimed to stress *what* the component does in an unambiguous fashion; this supports interoperability among developers and among technologies, provides some ground concepts to establish, declare or infer relationships among software components, and eases the reuse of existing software, giving developers a significant help in the early discovery phases.

4 KNOWLEDGE MODEL

The Knowledge Model of the SSP environment offers, at the current state of development, those concepts and relations which are necessary for providing a sufficiently detailed description of software entities and for modeling the functionalities which have been presented in the use-cases section.

Reference to past research work (Oberle et al., 2006) on modeling ontologies for describing software systems has been made by reusing concepts from these ontologies for describing common software entities like: *component*, *library* and *software license*.

Our framework is centered about the description of software objects, providing several semantic anchors through which they can be identified, classified according to different perspectives and needs, and thus easily retrieved on these same aspects.

SoftwareObject(s) can be mainly distinguished according to two different categories: Components, which are "Program modules that are designed to interoperate with each other at runtime", that is software objects for which there is a well-defined runtime behavior, and Library(ies) which define "collections of subprograms used to develop software".

Other classes offer further perspectives over which software objects registered in the SSP repository may be clustered and accessed: License has been introduced to describe the diverse software licenses adopted by software developers and vendors. This way users may filter their choice if, as an example, they need only software licensed under a specific contract. This filtering can even less explicit, by automatic reasoning over class of licenses and the relationships between them. A property licenseIncompatibleWith allows to establish incompatibilities between use of components licensed under different contracts, while the class LicenseStyle describes categories of licenses which share common aspects. A reification technique – see (Gangemi & Mika, 2003) for a wider discussion on this topic – has been adopted to describe license styles both as objects of the domain as well as classes of licenses (so, as rdfs:subClassOf License), still remaining inside a first order description of the domain. This way we can "talk about" software licenses as ground objects (which may exhibit specific contractual expressions, have a reference web site etc…) and, at the same time, consider them as set of licenses, offering class level restrictions on the values that their belonging instances should expose on their properties.

The explicit link between the objects (instances of LicenseStyle) and the set of Licenses (subclasses of License) is outside of the ontology vocabulary and is handled by the semantic repository, which automatically generates subclasses of License for each new introduced license style.

Specific Tasks can be defined in the repository, to help clustering components according to their purposes.

The same reification technique described above is used to automatically generate subcategories of SWObject which cluster sets of components and libraries according to their purposes.

5 ARCHITECTURE

The semantic repository publishes a set of REST API, in compliance to the well known architectural style described in (Fielding, 2000) allowing clients to easily consume its services, and enabling any kind of Web 2.0 buzzword-compliant mashup. The RESTlet framework was embedded into a servlet container to deploy the repository as a web application.

We also developed a REST Eclipse-based client consuming the repository's web services, decoupling the client-server interaction from the UI contributions.

The repository location can be both local (i.e. this can be achieved simply deploying the repository web application inside Eclipse itself, exploiting the embedded Jetty server used by the *help* plugin), or remote, and it can be chosen using the provided preference page, accessed in the usual Eclipse way.

Two views were implemented: the *Repository Explorer*, on the left, allows the developer to browse components by name, version, license, tags, tasks or navigate the semantic relations among the

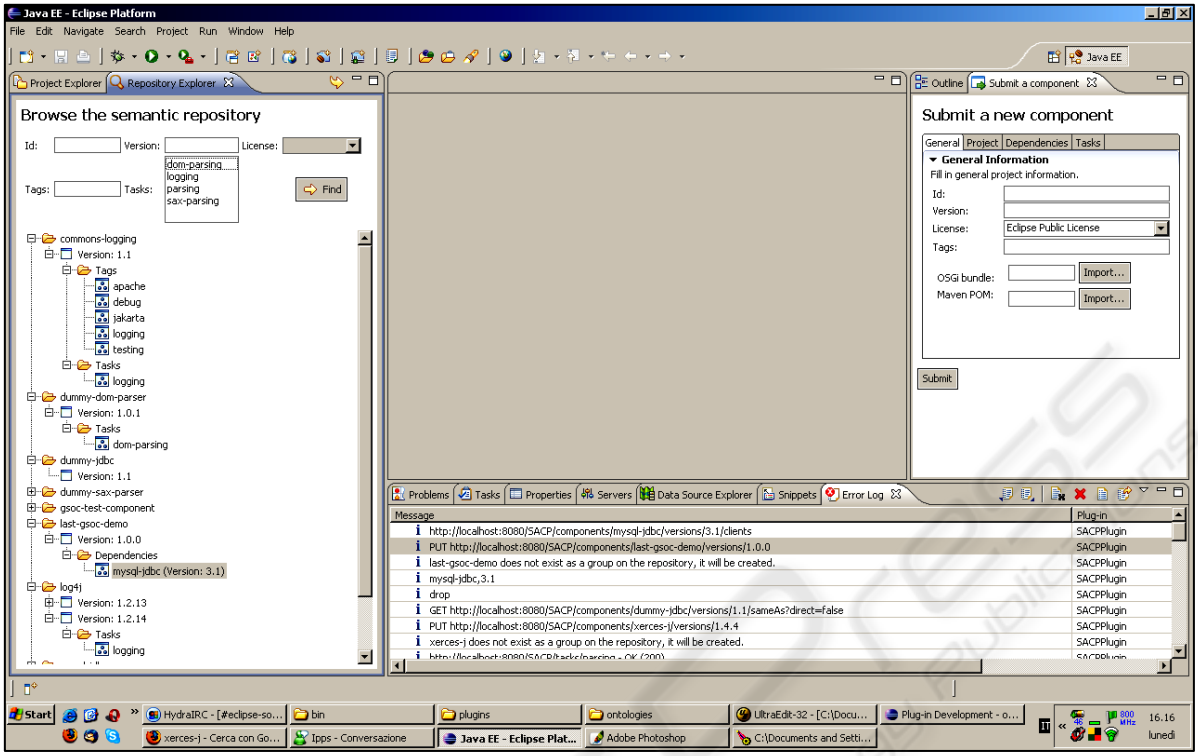

Figure 2: SSP Eclipse plug-in - UI Contribution.

components; the *Submit a new component* view makes use of the Eclipse SWT Forms widgets to provide developers with an elegant and fast way to submit a new component to the repository.

6 CONCLUSIONS AND FUTURE WORKS

In this paper we introduced a novel approach to software components and libraries discovery and provisioning. Indeed we believe current mainstream provisioning systems lack a shared vocabulary and technology-independent formalization of the software domain, supporting richer semantic description to support reasoning and the generation of a consensus based upon the specific domain the considered software belongs to.

Future iterations will involve a deeper axiomatization of *License* and *License-style* concepts, since they represent the contract between the product provider and the consumers, which often is a strict non-functional requirement to be satisfied when a third-party software is chosen. A strong investigation on "software specifications" could contribute to further discriminative arguments for facilitating classification (and thus more precise retrieval) of software objects in the repository. Integration with – and metadata reuse from – OSGi (http://www.osgi.org/) and Maven (http://maven.apache.org/), and user interface improvements are top priorities for the project.

REFERENCES

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* University of California Irvine, PhD Dissertation.

Gangemi, A., & Mika, P. (2003). Understanding the Semantic Web through Descriptions and Situations. *DOA/CoopIS/ODBASE.*

Oberle, D., Lamparter, S., Grimm, S., Vrandecic, D., Staab, S., & Gangemi, A. (2006). Towards Ontologies for Formalizing Modularization and Communication in Large Software Systems. *Journal of Applied Ontology , 1* (2), 163-202.

Sguera, S. (2007). Retrieved from http://code.google.com/soc/2007/eclipse/appinfo.html?csaid=1221666D7EBA3415.